CS 61A
Summer 2025

Efficiency, Scheme

Discussion 8: July 24, 2025

Scheme

Before working on the discussion problems, you can read over the introduction/refresher for Scheme below!

Atomic expressions (also called *atoms*) are expressions without sub-expressions, such as numbers, boolean values, and symbols.

```
scm> 1234    ; integer
1234
scm> 123.4   ; real number
123.4
scm> #f      ; the Scheme equivalent of False in Python
#f
```

A Scheme *symbol* is equivalent to a Python name. A symbol evaluates to the value bound to that symbol in the current environment. (They are called symbols rather than names because they include `+` and other arithmetic symbols.)

```
scm> quotient      ; A symbol bound to a built-in procedure
#[quotient]
scm> +             ; A symbol bound to a built-in procedure
#[+]
```

In Scheme, *all* values except `#f` (equivalent to `False` in Python) are true values (unlike Python, which has other false values, such as `0`).

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

All non-primitive expressions in Scheme are known as **combinations** and have the following syntax:

```
(<operator> <operand1> <operand2> ...)
```

Combinations are expressions that combine multiple expressions. Here, `<operator>`, `<operand1>`, and `<operand2>`. are all expressions. The number of operands depends on the operator. There are two types of combinations:

1. A **call expression**, whose operator evaluates to a procedure
2. A **special form expression**, whose operator is a special form

Scheme uses Polish prefix notation, in which the operator expression comes before the operand expressions. For example, to evaluate `3 * (4 + 2)`, we write:

```
scm> (* 3 (+ 4 2))
18
```

Just like in Python, to evaluate a call expression:

1. Evaluate the operator. It should evaluate to a procedure.
2. Evaluate the operands, left to right.
3. Apply the procedure to the evaluated operands.

Here are some examples using built-in procedures:

```
scm> (+ 1 2)
3
scm> (- 10 (/ 6 2))
7
scm> (modulo 35 4)
3
scm> (even? (quotient 45 2))
#t
```

The operator of a special form expression is a special form. What makes a special form "special" is that they do not follow the three rules of evaluation stated in the previous section. Instead, each special form follows its own special rules for execution, such as short-circuiting before evaluating all the operands.

Some examples of special forms that we'll study today are the `define`, `if`, `cond`, and `lambda` forms. Read their corresponding sections below to find out what their rules of evaluation are!

Define: The `define` form is used to assign values to symbols. It has the following syntax:

```
(define <symbol> <expression>)
```

```
scm> (define pi (+ 3 0.14))
pi
scm> pi
3.14
```

To evaluate the `define` expression:

1. Evaluate the final sub-expression (`<expression>`), which in this case evaluates to `3.14`.
2. Bind that value to the symbol (`symbol`), which in this case is `pi`.
3. Return the symbol.

The `define` form can also define new procedures, described in the "Defining Functions" section. The `define` form can create a procedure and give it a name:

```
(define (<symbol> <param1> <param2> ...) <body>)
```

For example, this is how we would define the `double` procedure:

```
scm> (define (double x) (* x 2))
double
scm> (double 3)
6
```

Here's an example with three arguments:

```
scm> (define (add-then-mul x y z)
        (* (+ x y) z))
scm> (add-then-mul 3 4 5)
35
```

When a `define` expression is evaluated, the following occurs: 1. Create a procedure with the given parameters and <body>. 2. Bind the procedure to the <symbol> in the current frame. 3. Return the <symbol>.

The following two expressions are equivalent:

```
scm> (define add (lambda (x y) (+ x y)))
add
scm> (define (add x y) (+ x y))
add
```

If Expressions: The `if` special form evaluates one of two expressions based on a predicate.

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

The rules for evaluating an `if` special form expression are as follows:

1. Evaluate the `<predicate>`.
2. If the `<predicate>` evaluates to a true value (anything but `#f`), evaluate and return the value of the `<if-true>` expression. Otherwise, evaluate and return the value of the `<if-false>` expression.

For example, this expression does not error and evaluates to 5, even though the sub-expression `(/ 1 (- x 3))` would error if evaluated.

```
scm> (define x 3)
x
scm> (if (> (- x 3) 0) (/ 1 (- x 3)) (+ x 2))
5
```

The `<if-false>` expression is optional.

```
scm> (if (= x 3) (print x))
3
```

Let's compare a Scheme `if` expression with a Python `if` statement:

- In Scheme:

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

- In Python:

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

The Scheme `if` expression evaluates to a number (either 1 or 2, depending on `x`). The Python statement does not evaluate to anything, and so the 1 and 2 cannot be used or accessed.

Another difference between the two is that it's possible to add more lines of code into the suites of the Python `if` statement, while a Scheme `if` expression expects just a single expression in each of the `<if-true>` and `<if-false>` positions.

One final difference is that in Scheme, you cannot write `elif` clauses.

Cond Expressions: The `cond` special form can include multiple predicates (like if/elif in Python):

```
(cond
    (<p1> <e1>)
    (<p2> <e2>)
    ...
    (<pn> <en>)
    (else <else-expression>))
```

The first expression in each clause is a predicate. The second expression in the clause is the return expression corresponding to its predicate. The `else` clause is optional; its `<else-expression>` is the return expression if none of the predicates are true.

The rules of evaluation are as follows:

1. Evaluate the predicates `<p1>`, `<p2>`, …, `<pn>` in order until one evaluates to a true value (anything but `#f`).
2. Evalaute and return the value of the return expression corresponding to the first predicate expression with a true value.
3. If none of the predicates evaluate to true values and there is an `else` clause, evaluate and return `<else-expression>`.

For example, this `cond` expression returns the nearest multiple of 3 to `x`:

```
scm> (define x 5)
x
scm> (cond ((= (modulo x 3) 0) x)
           ((= (modulo x 3) 1) (- x 1))
           ((= (modulo x 3) 2) (+ x 1)))
6
```

Let: The `let` special form allows you to create *local* bindings within Scheme. The let special form consists of two elements: a list of two element pairs, and a body expression. Each of the pairs contains a symbol and an expression to be bound to the symbol.

```
(let ((var-1 expr-1)
      (var-2 expr-2)
      ...
      (var-n expr-n))
      body-expr)
```

When evaluating a `let` expression, a new frame local to the `let` expression is created. In this frame, each variable is bound to the value of its corresponding expression *at the same time*. Then, the body expression is evaluated in this frame using the new bindings.

```
(let ((a 1)
      (b (* 2 3)))
     (+ a b)) ; This let expression will evaluate to 7
```

Let expressions allow us to simplify our code significantly. Consider the following implementation of `filter`:

```
(define (filter fn lst)
    (cond ((null? lst) nil)
          ((fn (car lst)) (cons (car lst) (filter fn (cdr lst))))
          (else (filter fn (cdr lst)))))
```

Now consider this alternate expression using let:

```
(define (filter fn lst)
    (if (null? lst)
        nil
        (let ((first (car lst))
              (rest (cdr lst)))
           (if (fn first)
               (cons first (filter fn rest))
               (filter fn rest)))))
```

Although there are more lines of code for filter, by assigning the `car` and `cdr` to the variables `first` and `rest`, the recursive calls are much cleaner.

let expressions also prevent us from evaluating an expression multiple times. For example, the following code will only print out `x` once, but without let we would print it twice.

```
(define (print-then-return x)
   (begin (print x) x))

(define (print-then-double x)
   (let ((value (print-then-return x)))
       (+ value value)))

(print-then-double (+ 1 1))
; 2
; 4
```

Lambdas: The `lambda` special form creates a procedure.

```
(lambda (<param1> <param2> ...) <body>)
```

This expression will create and return a procedure with the given formal parameters and body, similar to a `lambda` expression in Python.

```
scm> (lambda (x y) (+ x y))        ; Returns a lambda procedure, but doesn't assign it to
     a name
(lambda (x y) (+ x y))
scm> ((lambda (x y) (+ x y)) 3 4)  ; Create and call a lambda procedure in one line
7
```

Here are equivalent expressions in Python:

```
>>> lambda x, y: x + y
<function <lambda> at ...>
>>> (lambda x, y: x + y)(3, 4)
7
```

The `<body>` may contain multiple expressions. A scheme procedure returns the value of the last expression in its body.

Q1: WWSD: Call Expressions

What would Scheme display? As a reminder, the built-in `quotient` function performs floor division.

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

a

```
scm> a
```

3

```
scm> (define b (- (+ (* 3 3) 2) 1))
```

b

```
scm> (+ a b)
```

13

```
scm> (= (modulo b a) (quotient 5 3))
```

#t

Q2: WWSD: Special Forms

What would Scheme display?

```
scm> (if (or #t (/ 1 0)) 1 (/ 1 0))
```

1

```
scm> ((if (< 4 3) + -) 4 100)
```

-96

```
scm> (cond
        ((and (- 4 4) (not #t)) 1)
        ((and (or (< 9 (/ 100 10)) (/ 1 0)) #t) -1)
        (else (/ 1 0))
    )
```

-1

```
scm> (let (
            (a (- 3 2))
            (b (+ 5 7))
          )
        (* a b)
        (if (< (+ a b) b)
            (/ a b)
            (/ b a)
        )
    )
```

12

```
scm> (begin
        (if (even? (+ 2 4))
            (print (and 2 0 3))
            (/ 1 0)
        )
        (+ 2 2)
        (or 2 0 3)
    )
```

3

2

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Walkthrough video

Q3: Factorial

Write a function that returns the factorial of a number.

```
(define (factorial x)
  (if (< x 2)
      1
      (* x (factorial (- x 1))))
)

(expect (factorial 5) 120)
(expect (factorial 0) 1)
```

Walkthrough video

Q4: Perfect Fit

Definition: A perfect square is `k*k` for some integer `k`.

Implement `fit`, which takes non-negative integers `total` and `n`. It returns whether there are `n` **different** positive perfect squares that sum to `total`.

Important: Don't use the Scheme interpreter to tell you whether you've implemented it correctly. Discuss! On the final exam, you won't have an interpreter.

```
; Return whether there are n perfect squares with no repeats that sum to total

(define (fit total n)
    (define (f total n k)
        (if (and (= n 0) (= total 0))
            #t
        (if (< total (* k k))
            #f
        (or (f total n (+ k 1)) (f (- total (* k k)) (- n 1) (+ k 1)))
        )))
    (f total n 1))

(expect (fit 10 2) #t)  ; 1*1 + 3*3
(expect (fit 9 1)  #t)  ; 3*3
(expect (fit 9 2)  #f)  ;
(expect (fit 9 3)  #f)  ; 1*1 + 2*2 + 2*2 doesn't count because of repeated 2*2
(expect (fit 25 1)  #t) ; 5*5
(expect (fit 25 2)  #t) ; 3*3 + 4*4
```

Use the `(or _ _)` special form to combine two recursive calls: one that uses `k*k` in the sum and one that does not. The first should subtract `k*k` from `total` and subtract 1 from `n`; the other should leaves `total` and `n` unchanged.

Efficiency

Before working on the discussion problems, you can read over the introduction/refresher for efficiency below!

Throughout this class, we have mainly focused on *correctness* — whether a program produces the correct output. However, computer scientists are also interested in creating *efficient* solutions to problems. One way to quantify efficiency is to determine how a function's *runtime* changes as its input changes. In this class, we measure a function's runtime by the number of operations it performs.

A function `f(n)` has...

- constant runtime if the runtime of `f` does not depend on `n`. Its runtime is $\Theta(1)$.
- logarithmic runtime if the runtime of `f` is proportional to `log(n)`. Its runtime is $\Theta($`log(n)`$)$.
- linear runtime if the runtime of `f` is proportional to `n`. Its runtime is $\Theta($`n`$)$.
- quadratic runtime if the runtime of `f` is proportional to `n^2`. Its runtime is $\Theta($`n^2`$)$.
- exponential runtime if the runtime of `f` is proportional to `b^n`, for some constant `b`. Its runtime is $\Theta($`b^n`$)$.

Example 1: It takes a single multiplication operation to compute `square(1)`, and it takes a single multiplication operation to compute `square(100)`. In general, calling `square(n)` results in a *constant* number of operations that does not vary according to `n`. We say `square` has a runtime complexity of $\Theta(1)$.

input	function call	return value	operations
1	`square(1)`	1*1	1
2	`square(2)`	2*2	1
...	...	...	...
100	`square(100)`	100*100	1
...	...	...	...
n	`square(n)`	n*n	1

Example 2: It takes a single multiplication operation to compute `factorial(1)`, and it takes 100 multiplication operations to compute `factorial(100)`. As `n` increases, the runtime of `factorial` increases *linearly*. We say `factorial` has a runtime complexity of $\Theta($`n`$)$.

input	function call	return value	operations
1	`factorial(1)`	1*1	1
2	`factorial(2)`	2*1*1	2
...	...	...	...
100	`factorial(100)`	100*99*...*1*1	100
...	...	...	...
n	`factorial(n)`	n*(n-1)*...*1*1	n

Example 3: Consider the following function:

```
def bar(n):
    for a in range(n):
        for b in range(n):
            print(a,b)
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Evaulating `bar(1)` results in a single `print` call, while evalulating `bar(100)` results in 10,000 `print` calls. As `n` increases, the runtime of `bar` increases *quadratically*. We say `bar` has a runtime complexity of Θ(`n^2`).

input	function call	operations (prints)
1	`bar(1)`	1
2	`bar(2)`	4
...	...	...
100	`bar(100)`	10000
...	...	...
n	`bar(n)`	n^2

Example 4: Consider the following function:

```
def rec(n):
    if n == 0:
        return 1
    else:
        return rec(n - 1) + rec(n - 1)
```

Evaluating `rec(1)` results in a single addition operation. Evaluating `rec(4)` results in `2^4 - 1` = 15 addition operations, as shown by the diagram below.

During the evaulation of `rec(4)`, there are two calls to `rec(3)`, four calls to `rec(2)`, eight calls to `rec(1)`, and 16 calls to `rec(0)`.

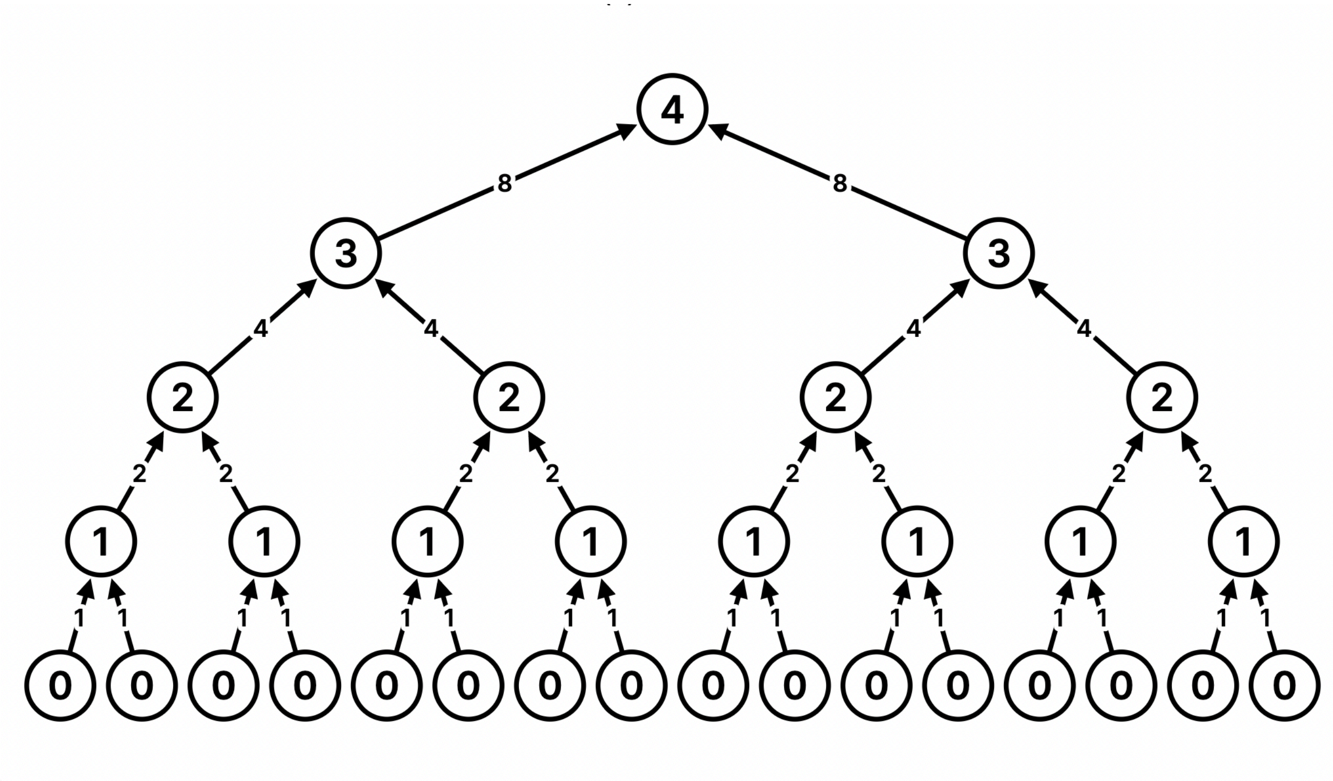
So we have eight instances of `rec(0) + rec(0)`, four instances of `rec(1) + rec(1)`, two instances of `rec(2) + rec (2)`, and a single instance of `rec(3) + rec(3)`, for a total of $1 + 2 + 4 + 8 = 15$ addition operations.

As `n` increases, the runtime of `rec` increases *exponentially*. In particular, the runtime of `rec` approximately doubles when we increase `n` by 1. We say `rec` has a runtime complexity of Θ(`2^n`).

input	function call	return value	operations
1	`rec(1)`	2	1
2	`rec(2)`	4	3
...	...	...	...
10	`rec(10)`	1024	1023
...	...	...	...
n	`rec(n)`	2^n	2^n - 1

Tips for finding the order of growth of a function's runtime:

- If the function is recursive, determine the number of recursive calls and the runtime of each recursive call.
- If the function is iterative, determine the number of inner loops and the runtime of each loop.
- Ignore coefficients. A function that performs `n` operations and a function that performs `100 * n` operations are both linear.
- Choose the largest order of growth. If the first part of a function has a linear runtime and the second part has a quadratic runtime, the overall function has a quadratic runtime.



Above: Call structure of rec(4).

- In this course, we only consider constant, logarithmic, linear, quadratic, and exponential runtimes.

Q5: WWPD: Orders of Growth

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

What is the *worst-case* runtime of `is_prime`?

```
def is_prime(n):
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

The worst-case runtime of `is_prime` occurs when `n` is actually prime. Each iteration of the for-loop takes constant time, and we have up to `n - 2` iterations. Therefore, the worst-case runtime of `is_prime` is linear.

What is the order of growth of the runtime of `bar(n)` with respect to `n`?

```
def bar(n):
    i, sum = 1, 0
    while i <= n:
        sum += biz(n)
        i += 1
    return sum

def biz(n):
    i, sum = 1, 0
    while i <= n:
        sum += i**3
        i += 1
    return sum
```

The while-loop in `bar` iterates for `n` loops, so `n` calls to `biz(n)` are made.

A single `biz(n)` call runs in linear time because the while-loop in `biz` iterates for `n` constant-time loops. Don't be confused by `i**3`: evaluating `i**3` takes constant time even though the result is cubic.

The runtime complexity of `bar` is Θ(n * n) = Θ(n^2) and the runtime is quadratic.

What is the order of growth of the runtime of `foo` in terms of `n`, where `n` is the length of `lst`? Assume that slicing a list and evaluating `len(lst)` take constant time.

Express your answer with Θ notation.

```
def foo(lst, i):
    mid = len(lst) // 2
    if mid == 0:
        return lst
    elif i > 0:
        return foo(lst[mid:], -1)
    else:
        return foo(lst[:mid], 1)
```

A `foo` call makes a single recursive call that halves the length of the argument for `lst`. We need approximately log(n) calls to reach the base case of a `lst` with length one or less.

The nonrecursive portion of each call takes constant time, so the overall runtime of `foo` is logarithmic and the runtime complexity of `foo` is $\Theta(\log(n))$.

Note: We made this problem easier by assuming that slicing a list takes constant time; in reality, slicing a list generally takes linear time with respect to the size of the slice.

Q6: Bonk

Describe the order of growth of the function below.

```
def bonk(n):
    sum = 0
    while n >= 2:
        sum += n
        n = n / 2
    return sum
```

Choose one of:

- Constant
- Logarithmic
- Linear
- Quadratic
- Exponential
- None of these

Logarithmic.

Explanation: As we increase the value of `n`, the amount of time needed to evaluate a call to `bonk` scales logarithmically. Let's use the number of iterations of our while loop to illustrate an example. When `n = 1`, our loop iterates 0 times. When `n = 2`, our loop iterates 1 time. When `n = 4`, we have 2 iterations. And when `n = 8`, a call to `bonk(8)` results in 3 iterations of this while loop. As the value of the input scales by a factor of 2, the number of iterations increases by 1. This indicates that this function runtime has a logarithmic order of growth.

You're done! Excellent work this week. Please be sure to fill out your TA's attendance form to get credit for this discussion!