

INSTRUCTIONS

- You have 3 hours to complete the exam.
- The exam is closed book, closed notes, closed computer, closed calculator, except three hand-written 8.5" × 11" crib sheet of your own creation and the official CS 61A midterm 1, midterm 2, and final study guides.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

Last name	
First name	
Student ID number	
CalCentral email ( _@berkeley.edu)	
TA	
Name of the person to your left	
Name of the person to your right	
<i>All the work on this exam is my own.</i> (please sign)	

POLICIES & CLARIFICATIONS

- If you need to use the restroom, bring your phone and exam to the front of the room.
- You may use built-in Python functions that do not require import, such as `min`, `max`, `pow`, `len`, `abs`, `sum`, `next`, `iter`, `list`, `tuple`, `map`, `filter`, `zip`, `all`, and `any`.
- You **may not** use example functions defined on your study guides unless a problem clearly states you can.
- For fill-in-the-blank coding problems, we will only grade work written in the provided blanks. You may only write one Python statement per blank line, and it must be indented to the level that the blank is indented.
- Unless otherwise specified, you are allowed to reference functions defined in previous parts of the same question.

### 1. (10 points) Iterators are inevitable

For each of the expressions in the table below, write the output displayed by the interactive Python interpreter when the expression is evaluated. The output may have multiple lines. The first row is completed for you.

- If an error occurs, write **ERROR**, but include all output displayed before the error.
- To display a function value, write **FUNCTION**.
- To display an iterator/generator value, write **ITERATOR**.
- If an expression would take forever to evaluate, write **FOREVER**.

The interactive interpreter displays the contents of the `repr` string of the value of a successfully evaluated expression, unless it is `None`.

Assume that you have started `python3` and executed the code shown on the left first, then you evaluate each expression on the right in the order shown. Expressions evaluated by the interpreter have a cumulative effect.

```
def love():
    yield 1000
    yield from [2000, 3000]
```

```
x = love()
L = list(x)
```

```
def alternate(real, ity):
    i1, i2 = iter(real), iter(ity)
    try:
        while True:
            yield next(i1)
            yield next(i2)
    except StopIteration:
        yield 'inevitable'
```

```
thanos = ['power', 'space', 'reality']
tony = ['mind', 'soul', 'time']
i = iter(tony)
next(i)
tony.extend(list(i))
thanos = tony[2::-2]
```

Expression	Output
<code>pow(10, 2)</code>	100
<code>print((print("end", print("game")), x))</code>	game end None None Iterator
<code>L</code>	[1000, 2000, 3000]
<code>next(x)</code>	Error
<code>tony</code>	['mind', 'soul', 'time', 'soul', 'time']
<code>list(alternate(thanos[1:], thanos))</code>	['mind', 'time', 'in- evitable']

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

**2. (10 points) Waitlisted**

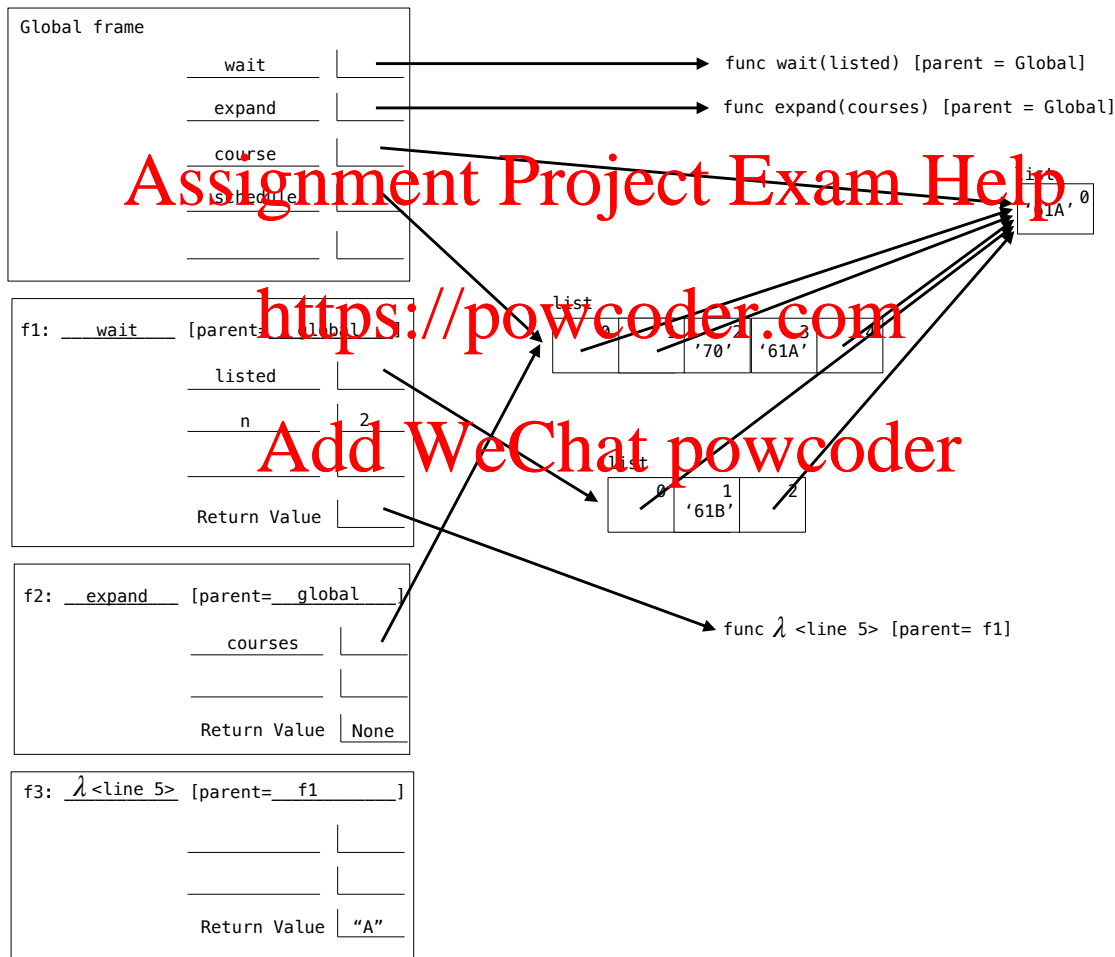
Fill in the environment diagram that results from executing the code on the right until the entire program is finished, an error occurs, or all frames are filled. *You may not need to use all of the spaces or frames.*

A complete answer will:

- Add all missing names and parent annotations to all local frames.
- Add all missing values created or referenced during execution.
- Show the return value for each local frame.
- Use box-and-pointer diagrams for lists and tuples.

```

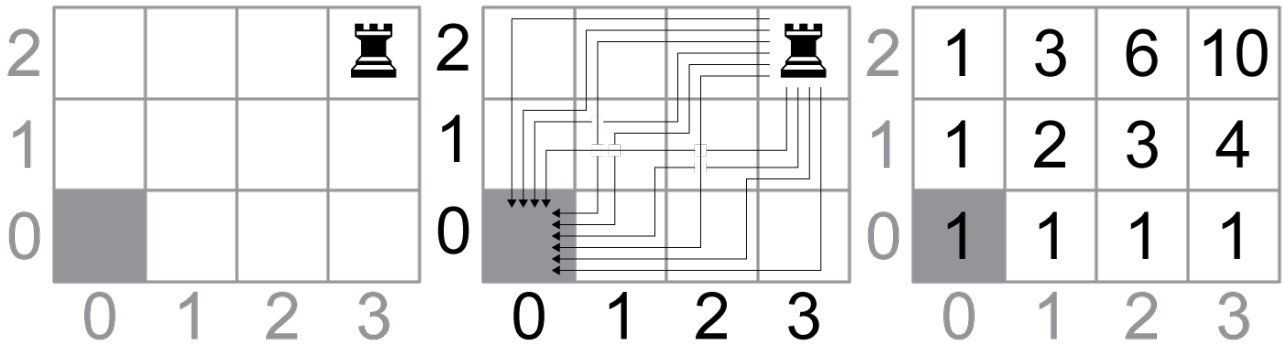
1 def wait(listed):
2   expand(schedule)
3   listed.insert(1, '61B')
4   n = sum([1 for c in listed if c is course])
5   return lambda: schedule[0][0][n]
6
7 def expand(courses):
8   courses.append('70')
9   courses.extend(course + [course])
10
11 course = ['61A']
12 schedule = [course, course]
13 wait(schedule[:])()
```



3. (9 points) I just want to go home!...

- (a) (3 pt) A rook is a piece in the game of chess that can move any number of squares vertically or horizontally. We put a rook somewhere on integer coordinates in the first quadrant ( $0 \leq x \leq \infty$ ,  $0 \leq y \leq \infty$ ) and put a spell on it so that it can only move toward the origin (i.e., either down or left).

Complete the function `paths2D(x, y)` to calculate how many different paths there are to get home at (0, 0) given a starting point (x, y). E.g., the rook at (3, 2) could get back to (0, 0) any one of 10 ways, and the number of paths for each starting square in ( $0 \leq x \leq 3$ ,  $0 \leq y \leq 2$ ) is shown below.



Assignment Project Exam Help

```
def paths2D(x, y):
    """
    >>> paths2D(3,2)
    10
    """
```

<https://powcoder.com>

Add WeChat powcoder

```
    if x == 0 or y == 0:
        return 1

    else:
        return paths2D(x - 1, y) + paths2D(x, y - 1)
```

- (b) (1 pt) Circle the  $\Theta$  expression that describes the running time of `paths2D(n, n)` as a function of  $n$ .

$\Theta(1)$

$\Theta(\log n)$

$\Theta(n)$

$\Theta(n^2)$

$\Theta(2^n)$

None of these

- (c) (5 pt) One of the elements of Abstraction is *generalization*! Why stop at 2D? That is, why not have a rook that can only move toward the origin in N-dimensions?

Complete the function `pathsND(vector)`, to return the number of paths home for this rook starting from `vector`, a list specifying its position in N-dimensions: `[x, y, z, ...]`.

```
def decrement_at(vector, i):
```

```
    vector_copy = vector[:]
```

```
    vector_copy[i] -= 1
```

```
    return vector_copy
```

```
def pathsND(vector):
```

```
    """
```

```
    >>> pathND([3, 2])
```

```
    10
```

```
    >>> pathND([3, 1, 2])
```

```
    60
```

```
    """
```

```
    if sum(vector) == 0:
```

```
        return 1
```

```
    else:
```

```
        return sum([pathsND(decrement_at(vector, i)) for i in range(len(vector)) if vector[i] > 0])
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

#### 4. (10 points) Streams

- (a) (6 pt) We provide `map-stream2` that calls `f` on each of the elements of streams `s1` and `s2`. Refer to `prefix` defined on the Final Study Guide. Fill in the blanks.

```
(define (map-stream2 f s1 s2)      ;; This allows us to call a 2-argument f
  (if (or (null? s1) (null? s2))  ;; Now we don't need add-streams since we can use this!
      nil
      (cons-stream (f (car s1) (car s2))
                    (map-stream2 f
                                (cdr-stream s1)
                                (cdr-stream s2))))))
```

```
scm> (define (spew x) (cons-stream x (spew x)))
spew
scm> (prefix (spew 3) 5)
```

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

```
scm> (define garply (cons-stream 1 (map-stream2 + (spew 1) garply)))
garply
scm> (prefix garply 5)
```

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

```
scm> (define strange (cons-stream nil (map-stream2 cons garply strange)))
strange
scm> (prefix strange 5)
```

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

- (b) (2 pt) We remind you of the definition of `map-stream`. Generate the stream `baz` using only calls to `cons-stream`, `map-stream` and/or `map-stream2`. You may add a `lambda` in there if needed.

```
(define (map-stream f s)
  (if (null? s)
      nil
      (cons-stream (f (car s))
                    (map-stream f (cdr-stream s)))))
```

```
scm> (define baz (cons-stream 1 (cons-stream 2 (map-stream (lambda (x) (+ x 1)) baz))))
baz
scm> (prefix baz 10)
(1 2 2 3 3 4 4 5 5 6)
```

- (c) (2 pt) Circle True or False

`cons-stream` is a special form:      True      False

`cdr-stream` is a special form:      True      False

**5. (6 points) Scope!**

For each of the following expressions, indicate what a lexically-scoped Scheme will return and what a dynamically-scoped Scheme will return. If the evaluation results in an error, just write the word error.

(Note: this is the first thing you type into the Scheme session)

```
scm> (define x 10)
x
scm> (define y 5)
y
scm> (define (f x) (* x y))
f
scm> (let ((y 20))
      (f 3))
```

15 in lexical scope, 60 in dynamic scope

```
scm> (define x 2)
x
scm> (define y 20)
y
scm> (define (foo x)
      (lambda () (* x y)))
foo
scm> ((foo 3) 5)
```

15 in lexical scope, 10 in dynamic scope

**6. (2 points) Potpourri**

What was NOT one of the “risks” mentioned in detail in Lecture 38 on Social Implications / Society?

- ☐ Computers and War (e.g., Autonomous weapons)
- ☐ Computers and Medicine (e.g., Therac-25)
- ☒ Computers and Elections (e.g., The 2016 election)
- ☐ Computers and Privacy (e.g., Ten Principles of Online Privacy)

What was NOT one of the technologies mentioned in detail in Lecture 37 on Distributed Computing?

- ☐ TOR (i.e., The Onion Router)
- ☐ Client / Server Architectures (e.g., CS61A’s website)
- ☒ IoT (i.e., Internet of Things)
- ☐ DNS (i.e., The Domain Name System)

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

### 7. (8 points) TCO!

- (a) (6 pt) The `count-evens` procedure takes a list of integers and returns the number of elements that are even. Rewrite `count-evens` as a tail-call optimized procedure by filling in the blanks below.

```
(define (count-evens ints)
  (cond ((null? ints) 0)
        ((even? (car ints)) (+ 1 (count-evens (cdr ints))))
        (else (count-evens (cdr ints)))))

(define (count-evens-tail ints)

  (define (helper ints total)

    (cond ((null? ints) total)

          ((even? (car ints)) (helper (cdr ints) (+ total 1)))

          (else (helper (cdr ints) total))))

  (helper ints 0))
```

- (b) (1 pt) Circle the  $\Theta$  expression that describes the running time of `count-evens-tail` where  $n$  is the length of the input list `ints`.

$\Theta(1)$ 
 $\Theta(\log n)$ 
 $\Theta(n)$ 
 $\Theta(n^2)$ 
 $\Theta(2^n)$ 
 None of these

- (c) (1 pt) Circle the  $\Theta$  expression that describes the space complexity of `count-evens-tail` where  $n$  is the length of the input list `ints`.

$\Theta(1)$ 
 $\Theta(\log n)$ 
 $\Theta(n)$ 
 $\Theta(n^2)$ 
 $\Theta(2^n)$ 
 None of these

### 8. (4 points) Macros

The `if` special form has been removed from scheme. Implement an `if` macro using only `and/or`:

```
(define-macro (if condition then else)

  `(or (and ,condition ,then) (and (not ,condition) ,else)))
```

```
scm> (if #t 1 (/ 1 0))
1
scm> (if #f 1 (/ 1 0))
Error
```



**9. (10 points) SQL**

Given the tables `users`, `sales`, and `products` answer the following questions.

```
CREATE TABLE users AS
  SELECT "Jon" AS name, 0 AS user_id UNION
  SELECT "Arya"      , 1      UNION
  SELECT "Sansa"     , 2      UNION
  SELECT "Daenerys"  , 3      UNION
  SELECT "Cersei"    , 4;

CREATE TABLE products AS
  SELECT 0 AS product_id, 15 AS price UNION
  SELECT 1      , 10      UNION
  SELECT 2      , 8        UNION
  SELECT 3      , 20;

CREATE TABLE sales AS
  SELECT 3 AS user_id, 2 AS product_id UNION
  SELECT 3      , 1      UNION
  SELECT 1      , 0      UNION
  SELECT 0      , 3      UNION
  SELECT 4      , 3      UNION
  SELECT 3      , 3      UNION
  SELECT 1      , 3      UNION
  SELECT 2      , 0;
```

**(a) (4 pt)**

```
CREATE TABLE t AS
  SELECT u.name
  FROM users AS u, sales AS s
  WHERE u.user_id = s.user_id
  GROUP BY u.user_id
  ORDER BY COUNT(*) DESC
  LIMIT 1;
```

What is the value of the one row in the table `t` and what does this value represent?

Daenerys

The person with the most number of purchases

- (b) (6 pt)** Create a table called `large_spenders` that contains the name and amount spent by everyone who spent at least \$25

```
CREATE TABLE large_spenders AS

  SELECT u.name AS name, SUM(price) AS amount_spent

  FROM users AS u, products AS p, sales AS s

  WHERE u.user_id = s.user_id AND p.product_id = s.product_id

  GROUP BY u.user_id

  HAVING SUM(price) >= 25;
```

**10. (6 points) Scheme**

Modify scheme so that it keeps track of the number of times each procedure is called inside the evaluator. You will also add the primitive `call-count` that takes a procedure as its argument and returns the number of times the procedure has been called since the evaluator was started. This feature should work for both primitive and compound procedures. For example:

```
scm> (define (foo x)
      (* 2 (+ 1 (* 2 x))))
scm> (call-count foo)
0
scm> (call-count *)
0
scm> (foo 2)
10
scm> (call-count foo)
1
scm> (foo 10)
42
scm> (call-count foo)
2
scm> (call-count *)
4
scm> (call-count (lambda (x) x))
0
```

Your job is to modify the interpreter to make this work. We have provided several possibly relevant functions on the following pages. (You might not need all the lines!)

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder

```
CC = {}

def scheme_call_count(procedure):
    if procedure in CC:
        return CC[procedure]

    return 0

def scheme_apply(procedure, args, env):
    """Apply Scheme PROCEDURE to argument values ARGS (a Scheme list) in
    environment ENV."""

    check_procedure(procedure)

    if procedure in CC:
        CC[procedure] += 1

    else:
        CC[procedure] = 1

    if isinstance(procedure, BuiltinProcedure):
        return procedure.apply(args, env)

    else:
        new_env = procedure.make_call_frame(args, env)
        return eval_all(procedure.body, new_env)

def create_global_frame():
    """Initialize and return a single-frame environment with built-in names."""

    env = Frame(None)

    env.define('call-count', BuiltinProcedure(scheme_call_count))

    env.define('apply', BuiltinProcedure(scheme_apply))

    env.define('map', BuiltinProcedure(scheme_map))

    ...

    return env
```

Assignment Project Exam Help

<https://powcoder.com>

Add WeChat powcoder