CS 61A Structure and Interpretation of Computer Programs
Summer 2018

Final Solutions

INSTRUCTIONS

- You have 3 hours to complete the exam individually.
- The exam is closed book, closed notes, closed computer, and closed calculator, except for two hand-written 8.5" × 11" crib sheets of your own creation.
- Mark your answers **on the exam itself**. We will *not* grade answers written on scratch paper.

Last (Family) Name	
First (Given) Name	
Student ID Number	
Berkeley Email	
Teaching Assistant	○ Alex Stennet ○ Alex Wang ○ Cameron Malloy ○ Chae Park ○ Chris Allsman ○ Christina Zhang ○ Derek Wan ○ Erica Kong ○ Griffin Prechter ○ Jemin Desai ○ Jennifer Tsui ○ Jenny Wang ○ Kevin Li ○ Nancy Shaw
Exam Room and Seat	
Name of the person to your left	
Name of the person to your right	
All the work on this exam is my own. **(please sign)**	

POLICIES & CLARIFICATIONS

- You may use built-in Python functions that do not require import, such as `min`, `max`, `pow`, and `abs`.
- 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. (11 points) Jennifer and Chae's Cat Cafe

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. Each expression has at least one line of output.

- If an error occurs, write **Error**, but include all output displayed before the error.
- To display a function value, write **Function**.
- If an expression would take forever to evaluate, write **Forever**.

Assume that you have started `python3` (not `ipython` or other variants) and executed the code shown on the left first, then you evaluate each expression on the right in the order shown.

```
class Animal:
    def __init__(self, parent):
        self.parent = parent
        is_alive = False
    def __repr__(self):
        return 'cookie'
    def __str__(self):
        return 'I am an animal'

class Cat(Animal):
    is_alive = True

    def meow(self):
        def meower(self):
            if self.is_alive:
                return 'meower'
            return 'I am a ghost!'
        print('meow')
        return meower

    def curiosity(self, cat):
        print('adventure')
        cat.is_alive = not self.is_alive
        return cat

class CalicoCat(Cat):
    def __repr__(self):
        return 'brownie'

    def meow(self):
        print('purr')
        return Cat.meow(self)

rachel = Cat(None)
aaron = CalicoCat(rachel)
amy = Cat(None)
```

Expression	**Interactive Output**
`print(4, 5) + 1`	4 5 **Error**
`print(aaron)`	I am an animal
`CalicoCat.meow(rachel)`	purr meow **Function**
`Animal.meow(aaron)`	**Error**
`aaron.parent.curiosity(amy)`	adventure cookie
`Cat.meow = rachel.meow()`	meow
`amy.meow()`	'I am a ghost!'

2. (8 points) Jemin Watches Some Soccer

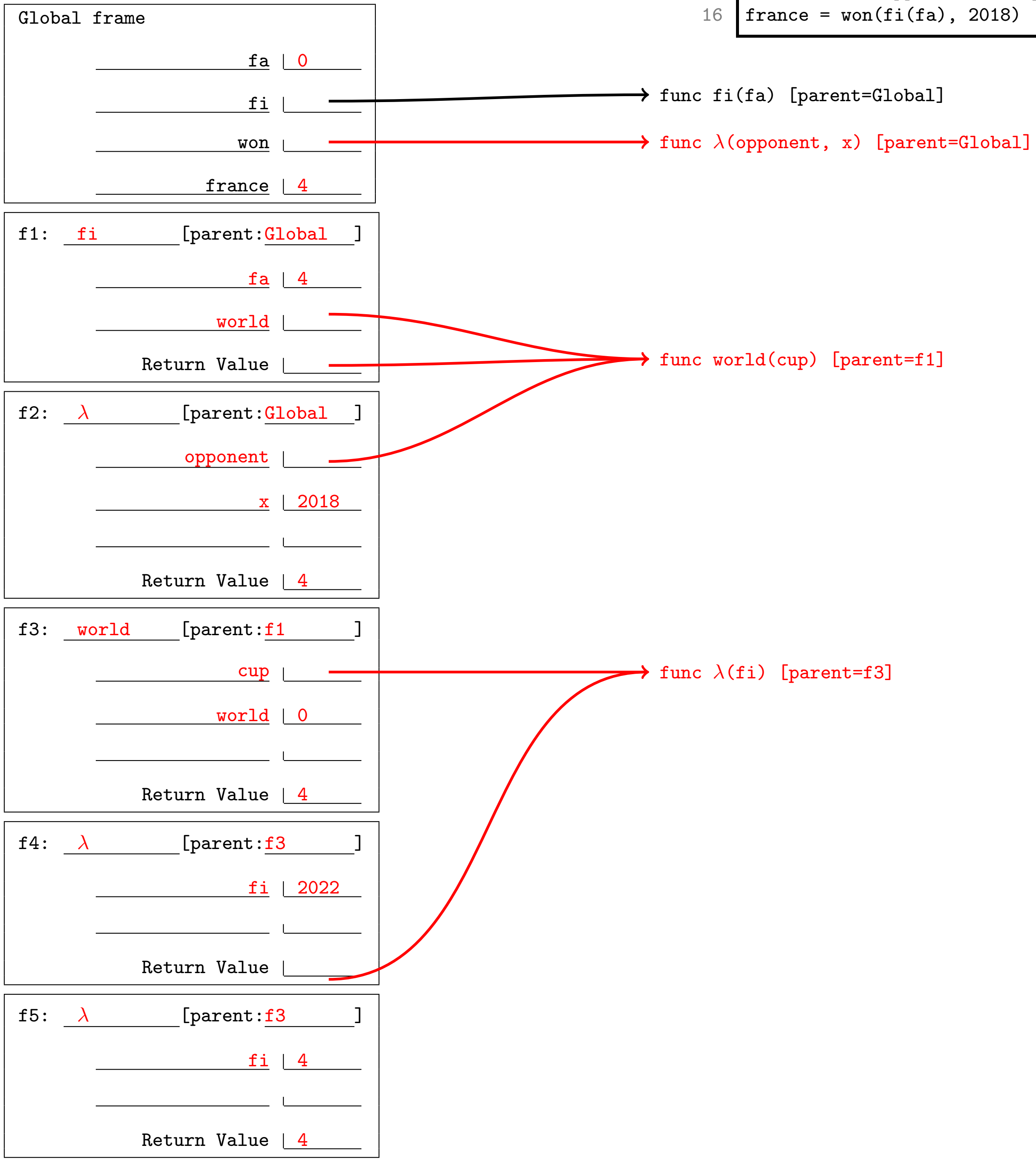
```
fa = 0

def fi(fa):
    def world(cup):
        nonlocal fa
        fa = lambda fi: world or fa or fi
        world = 0
        if (not cup) or fa:
            fa(2022)
            fa, cup = world + 4, fa
            return cup(fa)
        return fa(cup)
    return world

won = lambda opponent, x: opponent(x)
france = won(fi(fa), 2018)
```

Fill in the environment diagram that results from executing the code to 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.



```
Global frame
              fa | 0
              fi |  --> func fi(fa) [parent=Global]
             won |  --> func λ(opponent, x) [parent=Global]
          france | 4

f1: fi            [parent:Global]
              fa | 4
           world |  --> func world(cup) [parent=f1]
    Return Value |  --> func world(cup) [parent=f1]

f2: λ             [parent:Global]
        opponent |  --> func world(cup) [parent=f1]
               x | 2018
                 |
    Return Value | 4

f3: world         [parent:f1]
             cup |  --> func λ(fi) [parent=f3]
           world | 0
                 |
    Return Value | 4

f4: λ             [parent:f3]
              fi | 2022
                 |
    Return Value |  --> func λ(fi) [parent=f3]

f5: λ             [parent:f3]
              fi | 4
                 |
    Return Value | 4
```

3. (11 points) While You Evaluate These Function Calls-man, Don't Make An Error-ca

(a) (8 pt) For each of the expressions in the table below, write the output displayed by the interactive Scheme interpreter when the expression is evaluated. The output may have multiple lines. Each expression has at least one line of output.

- If an error occurs, write **Error**, but include all output displayed before the error.
- To display a procedure value, write **Procedure**.
- If an expression would take forever to evaluate, write **Forever**.

Assume that you have executed the code shown on the left first, then you evaluate each expression on the right in the order shown.

```
(define b 7)

(define hermish '(1 (* 2 3) b (+ 4 5)))

(define (shide x y)
  (lambda (z) (print z) (if x y z)))

(define jericho
  (lambda (a) (set! b a) (+ a b)))

(define asli hermish)

(define (jacob lst)
  (cond
    ((null? lst) lst)
    ((eq? (car lst) 5)
      (print 'hello)
      (list (+ 1 2)))
    (else
      (set-car! lst (eval (car lst)))
      (jacob (cdr lst)))))
```

Expression	**Interactive Output**
`(+ (length '(3 4)) 1)`	3
`((shide 0 b) 'amy)`	amy 7
`(jericho 5)`	10
`(jacob hermish)`	()
`asli`	(1 6 5 9)

(b) **(3 pt)** Draw a box-and-pointer diagram for the state of the Scheme pairs after executing the block of code below. Please erase or cross out any boxes or pointers that are not part of a final diagram. This code does not error. We've provided the diagram for `ex` as an example. The built-in procedure `length` returns the length of a Scheme list.

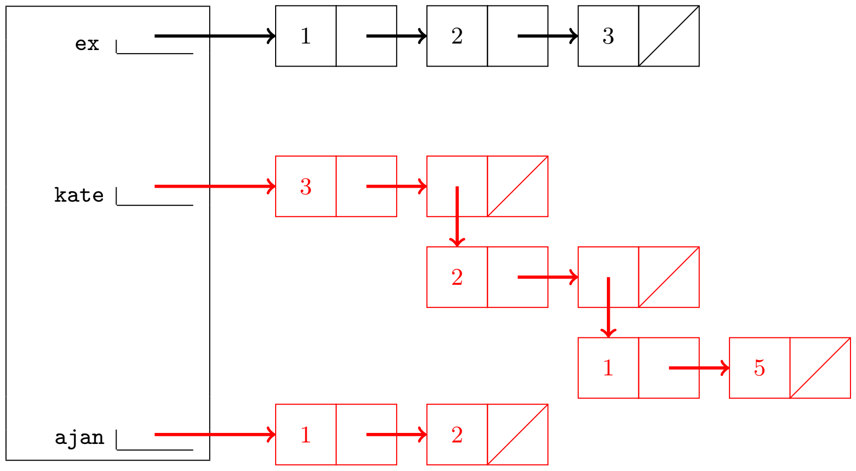
```
(define ex '(1 2 3))

(define (f x)
  (if (= x 0)
      5
      (list x (f (- x 1)))))

(define kate (f 3))

(define (g x)
  (if (list? x) (length x) x))

(define ajan (map g '(1 (2 (3)))))
```



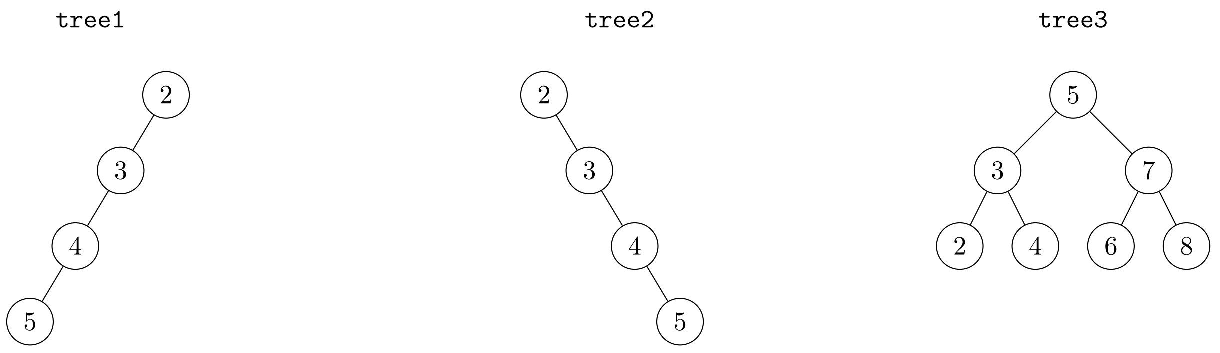
4. (3 points) BSTs

The following questions reference the BST class and diagrams provided below.

```
class BST:

    # Other methods not shown

    def __contains__(self, e):
        if self.label == e:
            return True
        elif e > self.label and self.right is not BST.empty:
            return e in self.right
        elif self.left is not BST.empty:
            return e in self.left
        return False
```



(a) (1 pt) Of `tree1` and `tree2`, which are valid BSTs?

○ Both ○ `tree1` only [✓ `tree2` only] ○ Neither

(b) (1 pt) What is the runtime of the `__contains__` method of the `BST` class with respect to n, the number of nodes in `tree3`?

○ $\Theta(1)$ [✓ $\Theta(\log n)$] ○ $\Theta(\sqrt{n})$ ○ $\Theta(n)$ ○ $\Theta(n^2)$ ○ $\Theta(2^n)$

(c) (1 pt) What is the runtime of the `__contains__` method of the `BST` class with respect to h, the height of `tree3`?

○ $\Theta(1)$ ○ $\Theta(\log h)$ ○ $\Theta(\sqrt{h})$ [✓ $\Theta(h)$] ○ $\Theta(h^2)$ ○ $\Theta(2^h)$

5. (9 points) Kevin and Griffin's Lunch Order

Kevin and Griffin are getting lunch before their sections. They each want to buy a main item, a snack, and a soft drink while staying within their budget.

(a) (2 pt) We represent the various lunch items with the `Food` class and its subclasses. There's a sale on snacks right now, so all snacks cost 40% less than their listed price. Berkeley charges a 5% tax on soft drinks, so those cost more than their base cost. Fill in the `cost` method for the `Snack` and `SoftDrink` classes.

For full credit, you must not hard-code the snack discount or the soda tax, in case they change in the future.

```
class Food:
    def __init__(self, name, base_cost):
        self.name = name
        self.base_cost = base_cost

    def cost(self):
        return self.base_cost


class Main(Food):
    type = "main"


class Snack(Food):
    type = "snack"
    discount = 0.4

    def cost(self):
        """
        >>> chips = Snack("chips", 1)
        >>> chips.cost()
        0.6
        """
        return self.base_cost * (1 - Snack.discount)


class SoftDrink(Food):
    type = "softdrink"
    sugar_tax = 0.05

    def cost(self):
        """
        >>> cola = SoftDrink("cola", 2)
        >>> cola.cost()
        2.1
        """
        return self.base_cost * (1 + SoftDrink.sugar_tax)
```

(b) **(5 pt)** Write `three_sum_budget`, which takes in three nonempty lists of positive numbers and a number `n`. This function should return the maximum sum less than or equal to `n` of one element from each of `lst1`, `lst2`, and `lst3`. If staying less than or equal to `n` is not possible, return 0.

```
def three_sum_budget(lst1, lst2, lst3, n):
    """Find the maximum sum <= n of one element from each of lst1, lst2, and lst3.
    >>> three_sum_budget([1, 2, 3], [6, 8, 10], [4], 100)
    17
    >>> three_sum_budget([1, 2, 4], [6, 8, 10], [2], 15)
    14
    >>> three_sum_budget([1, 2, 3], [4, 5, 6], [1, 2, 4], 6)
    6
    """
    def helper(lst1, lst2, lst3, total):
        if total > n:
            return 0
        elif lst1 == "done":
            return total
        options = []
        for item in lst1:
            options.append(helper(lst2, lst3, "done", total + item))
        return max(options)

    return helper(lst1, lst2, lst3, 0)
```

(c) **(2 pt)** Now let's put it all together. Implement `lunch_cost`, which takes in a list of foods (each one is either a `Main`, a `Snack`, or a `SoftDrink`) and a budget. Return the maximum you'll spend if you buy one of each item and your total cost does not exceed your budget. Use the functions and classes you wrote in parts (a) and (b).

```
def lunch_cost(foods, budget):
    """
    >>> lobster = Main('Lobster Tail', 25)
    >>> hotdog = Main('Hotdog', 5)
    >>> cider = SoftDrink('Sparkling Cider', 10)
    >>> cola = SoftDrink('Cola', 3)
    >>> fries = Snack('French Fries', 3)
    >>> foods = [lobster, hotdog, cider, cola, fries]
    >>> lunch_cost(foods, 100)
    37.3
    >>> lunch_cost(foods, 25)
    17.3
    """
    def costs_by_type(type):
        return [food.cost() for food in foods if food.type == type]

    mains = costs_by_type("main")
    snacks = costs_by_type("snack")
    drinks = costs_by_type("softdrink")
    return three_sum_budget(mains, snacks, drinks, budget)
```

6. (8 points) Christreena Finds Longer Paths

On HW 5, you wrote `long_paths`, which found all paths of a certain length that extend from the root to a leaf. Now, write `longer_paths`, which removes the restrictions that paths must begin at the root and end at a leaf.

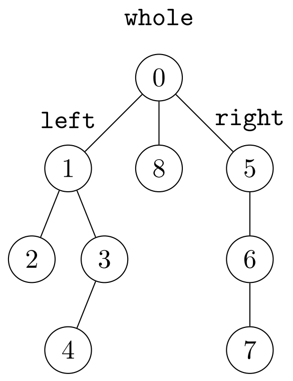
The length of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). A path may begin and end at any node. Paths must always go from one node to one of its branches; they may not go upwards. You do not need to worry about the order of the different paths.

The `Tree` class is provided below.

```
class Tree:
    def __init__(self, label, branches=[]):          def is_leaf(self):
        self.label = label                               return not self.branches
        self.branches = list(branches)
```

```
def longer_paths(t, n):
    """Return a list of all paths in T with length at least N. Note that a path
    may begin at any node and end at any node. Each subsequent element must be
    from a label of a branch of the previous value's node.
    >>> left = Tree(1, [Tree(2), Tree(3, [Tree(4), Tree(5)])])
    >>> right = Tree(6, [Tree(7, [Tree(8)])])
    >>> whole = Tree(0, [left, Tree(9), right])
    >>> for path in longer_paths(whole, 2):
    ...     print(path)
    ...
    [0, 1, 2]
    [0, 1, 3]
    [0, 1, 3, 4]
    [0, 1, 3, 5]
    [1, 3, 4]
    [1, 3, 5]
    [0, 6, 7]
    [0, 6, 7, 8]
    [6, 7, 8]
    """
    def helper(t, n, can_start_path):
        paths = []
        if n <= 0:
            paths.append([t.label])
        for b in t.branches:
            for subpath in helper(b, n - 1, False):
                paths.append([t.label] + subpath)
            if can_start_path:
                paths.extend(helper(b, n, True))
        return paths

    return helper(t, n, True)
```



7. (7 points) Streams and Jennyrators

(a) (1 pt) Write `generate_constant`, a generator function that repeatedly yields the same value forever.

```
def generate_constant(x):
    """A generator function that repeats the same value X forever.
    >>> two = generate_constant(2)
    >>> next(two)
    2
    >>> next(two)
    2
    >>> sum([next(two) for _ in range(100)])
    200
    """
    while True:
        yield x
```

(b) (3 pt) Now implement `black_hole`, a generator that yields items in `seq` until one of them matches `trap`, in which case that value should be repeated yielded forever. You may assume that `generate_constant` works. You **may not** index into or slice `seq`.

```
def black_hole(seq, trap):
    """A generator that yields items in SEQ until one of them matches TRAP, in
    which case that value should be repeatedly yielded forever.
    >>> trapped = black_hole([1, 2, 3], 2)
    >>> [next(trapped) for _ in range(6)]
    [1, 2, 2, 2, 2, 2]
    >>> list(black_hole(range(5), 7))
    [0, 1, 2, 3, 4]
    """
    for item in seq:
        if item == trap:
            yield from generate_constant(trap)
        else:
            yield item
```

(c) **(3 pt)** Now let's implement this in Scheme using streams. `black-hole` takes in an infinite stream of numbers and a value `trap`. It should return a stream that contains the items of `stream` until one of its elements matches `trap`, in which case the stream should repeat that value forever.

```
scm> (define (prefix s k) (if (= k 0) nil (cons (car s) (prefix (cdr-stream s) (- k 1)))))
prefix
scm> (define (naturals start) (cons-stream start (naturals (+ start 1))))
naturals
scm> (prefix (black-hole (naturals 1) 3) 8)
(1 2 3 3 3 3 3 3)
scm> (prefix (black-hole (naturals 5) 3) 5)
(5 6 7 8 9)
```

```
(define (black-hole stream trap)
  (cons-stream
    (car stream)
    (if (equal? (car stream) trap)
        (black-hole stream trap)
        (black-hole (cdr-stream stream) trap))))
```

8. (10 points) Nan-scheme Writes Cam-acros

In Python, we can do arithmetic using *infix* notation, where the operator goes between two operands, e.g. `3 + 4`. In Scheme, we have to use *prefix* notation for all call expressions, e.g. `(+ 3 4)`.

Let's add support for infix notation in Scheme!

(a) **(2 pt)** First, write the helper function `skip`, which skips the first `n` items in a list, returning the rest. For full credit, your solution must be tail recursive. You may assume that `n` is non-negative.

```
scm> (skip 2 '(1 2 3 4))
(3 4)

(define (skip n lst)
  (if (or (= n 0) (null? lst))
      lst
      (skip (- n 1) (cdr lst))))
```

(b) **(6 pt)** Now let's write `infix`, which takes in a list containing some arithmetic in infix notation and evaluates it. You only need to support addition and multiplication, but you do need to take the order of operations and parentheses into account. You may use `skip`, as well as `cadr` and `caddr`.

```
scm> (infix '(5))                scm> (infix '(2 + 3))          scm> (infix '(2 * (3 + 6)))
5                                5                              18
scm> (infix '(2 * 3))            scm> (infix '(2 * 3 + 6))      scm> (infix '(2 + 3 * 6))
6                                12                             20
```

```
1  (define (infix expr)
2    (cond
3      ((not (pair? expr)) expr) ; a single value
4      ((or (equal? (car expr) '*) (equal? (car expr) '+)) (eval expr)) ; already in prefix form

5      ((null? (cdr expr)) (infix (car expr)))

6      (else
7        (define left (infix (car expr)))


8        (define right (infix (caddr expr)))


9        (define operator (cadr expr))

10       (cond
11         ((equal? operator '+) (+ left (infix (skip 2 expr))))


12         ((equal? operator '*) (infix (cons (* left right)


13                                             (skip 3 expr))))))
```

(c) **(2 pt)** `infix` is great, but it only works on number literals. If we try to reference names, it errors.

```
scm> (define x 4)
x
scm> (infix '(x + 3))
Error: x is not a number
```

We can fix this by making a macro instead. Let's say we define `infix-macro` as:

```
(define-macro (infix-macro . expr) (infix expr))
```

Unfortunately, this doesn't quite work. What changes would need to be made to the code in part (b) so that `infix-macro` works like the tests below?

```
scm> (infix-macro x + 3)                         scm> (infix-macro 4 + (x + 3) * 5)
7                                                39
```

Please describe the specific changes you'd make and why you'd make them, mentioning line numbers.

Change `(eval expr)` in Line 4 to be `expr`. Change the blank in Line 12 from `(* left right)` to `` `(* ,left ,right)``. Likewise, change the addition in Line 11 to be `` `(+ ,left ,(infix (skip 2 expr)))``. We do this because we want macros to return code that can then be evaluated in the calling environment.

One point will be awarded for any answer that mentions that we need `infix` to return code, since macros need to return code.

The other point will be awarded if at least two of Lines 4, 11, and 12 has the correct new code (or at least correct relative to what the student actually wrote in part b).

9. (6 points) Birthday Query Language

Tiffany's birthday is coming up and the CS 61A staff wants to throw her a party! She's put the times she's available in a SQL table called `party_times`. The party will last 2 hours. Unfortunately, all of the times Tiffany is available conflict with some staff member's section. Each staff member's section is listed in `sections`.

The `time` column of both tables refers to the number of hours after noon that a section or party starts at. The `length` column refers to the length of a section in hours. The tables below are not complete.

party_times
time
2
4
5.5
7
...

sections		
staff_member	time	length
"Daniel"	4.5	1
"Jemmy"	3	1
"Lauren"	2.5	1
"Wenyuan"	1	2
...	...	...

(a) **(3 pt)** First, let's make a table called `available` which has a row for every combination of a staff member and a time slot for which that staff member is available. Note that a staff member can be available for multiple times and there can be multiple staff members available for a given time.

A staff member is available for a timeslot if their section does not conflict with any part of it. If their section ends at the same time the party starts (or vice versa), the staff member can still attend the party. You may assume that each staff member only has 1 section on the day of the party.

For example, Lauren could attend a party at 4, since her section ends at 3.5, but could not attend a party at 2, since her section would overlap with it.

```
CREATE TABLE available AS
  SELECT staff_member AS staff_member, party_times.time AS time
    FROM sections, party_times
    WHERE sections.time >= party_times.time + 2 OR
          sections.time + sections.length <= party_times.time;
```

(b) **(1 pt)** Uh, oh! Tiffany can no longer make the party time starting at 2. Write a single SQL statement that will mutate the `available` table to remove any availabilities listed for this time.

```
DELETE FROM available WHERE time = 2;
```

(c) **(2 pt)** Now let's find out what time the most staff members can make. Create a table called `best_times` that lists each party time and the number of staff members that can make it in descending order.

```
CREATE TABLE best_times
  SELECT time, count(*) FROM available
  GROUP BY time ORDER BY count(*) DESC;
```

10. (2 points) Alex-tra Lectures

There are three problems here: one for each of the extra lectures. Each problem is worth 1 points, but you can only earn a maximum of 2 points on this problem, so you only need to know two answers.

(a) (1 pt) Logic Programming

Why are logic programming languages (like Logic or Prolog) less efficient than SQL?
Limit your response to 15 words or less.

They allow arbitrary relations while SQL requires data and queries to fit a particular structure.

(b) (1 pt) Dynamic Programming

What is the main goal of memoization and dynamic programming? Limit your response to 10 words or less.

Do not perform repeated work

(c) (1 pt) Natural Language Processing

What does a leaf represent in a natural language syntax tree?

[✓ a single word] ○ a noun phrase ○ a verb phrase ○ a subordinate clause ○ a sentence

11. (0 points) Perfectly Balanced, As All Things Should Be

In this extra credit problem, you may choose one of the four instructors for the course. Your goal as a class is to evenly distribute your selections across the four options. If each instructor receives at least 20% of the votes, then everyone who properly marked an instructor for this problem will receive *one* (1) extra credit point.

You will not receive extra credit if you leave this problem blank, mark more than one bubble, or your selection is not clear.

○ James Uejio

○ Jen Thakar

○ Mitas Ray

○ Tammy Nguyen

12. (0 points) Wan More Thing

Thank you all for a fantastic summer!

We've hidden the names of all the instructors, TAs, and tutors somewhere within the exam. Can you find them all?

This isn't worth extra credit or anything. Obviously don't do this until you've finished the exam.

No more questions.