CS 61A Fall 2021

Structure and Interpretation of Computer Programs

Final

INSTRUCTIONS

This is your exam. Complete it either at exam.cs61a.org or, if that doesn't work, by emailing course staff with your solutions before the exam deadline.

This exam is intended for the student with email address `<EMAILADDRESS>`. If this is not your email address, notify course staff immediately, as each exam is different. Do not distribute this exam PDF even after the exam ends, as some students may be taking the exam in a different time zone.

For questions with **circular bubbles**, you should select exactly *one* choice.

- ◯ You must choose either this option
- ◯ Or this one, but not both!

For questions with **square checkboxes**, you may select *multiple* choices.

- ☐ You could select this choice.
- ☐ You could select this one too!

You may start your exam now. Your exam is due at <DEADLINE> Pacific Time. Go to the next page to begin.

Preliminaries

You can complete and submit these questions before the exam starts.

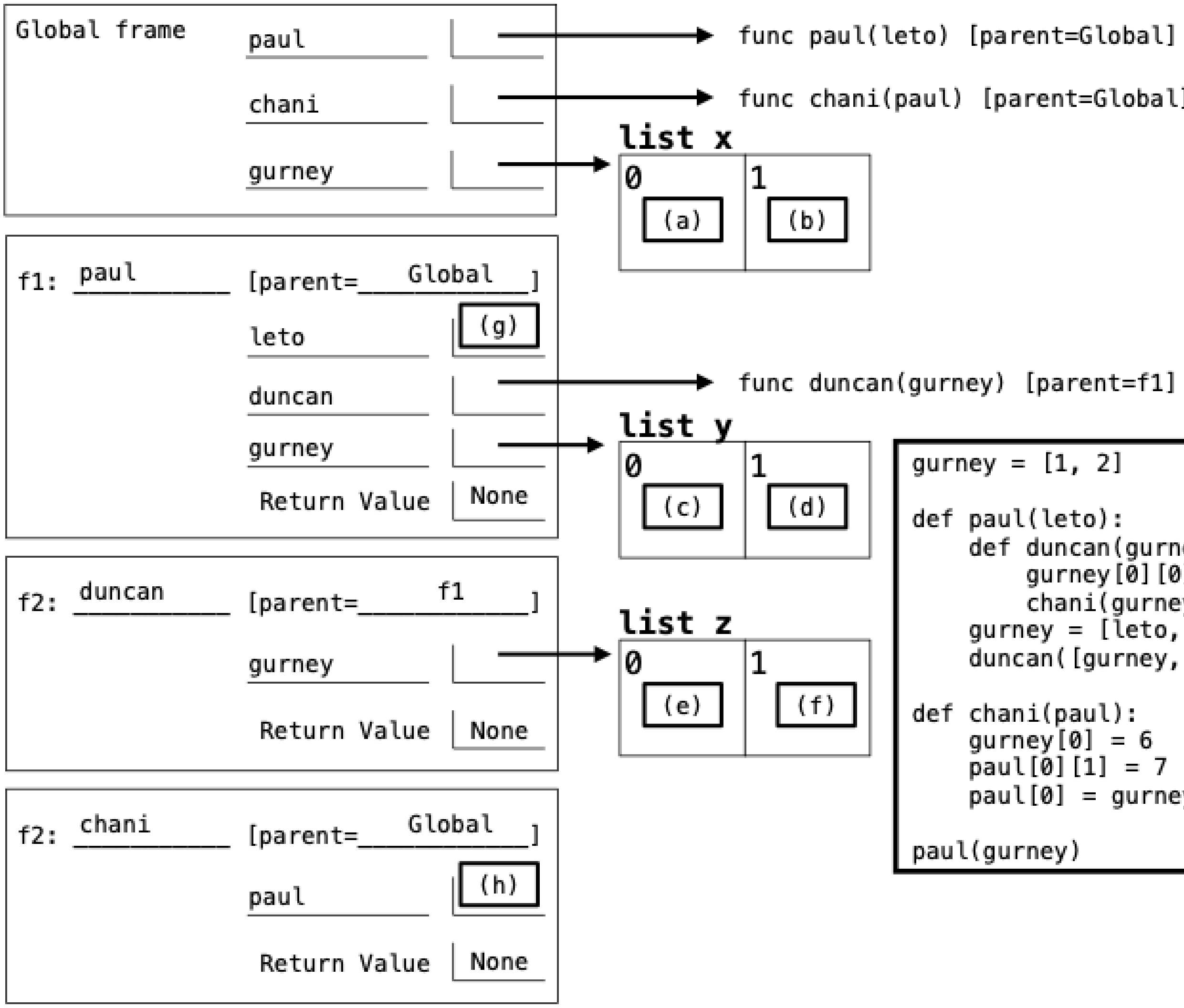
(a) What is your full name?

(b) What is your student ID number? A regex restricts inputs to numerical responses only.

1. (14.0 points) House Atreides

(a) (8.0 points)

The environment diagram below was generated by code that is provided to the right of the diagram.




```
gurney = [1, 2]

def paul(leto):
    def duncan(gurney):
        gurney[0][0] = 3
        chani(gurney)
    gurney = [leto, 4]
    duncan([gurney, 5])

def chani(paul):
    gurney[0] = 6
    paul[0][1] = 7
    paul[0] = gurney

paul(gurney)
```

i. (1.0 pt) Which of these could fill in blank (a)?

- ◯ 1
- ◯ 3
- ◯ 6
- ◯ 7

ii. (1.0 pt) Which of these could fill in blank (b)?

- ◯ 2
- ◯ 6
- ◯ 7
- ◯ an arrow to list y (named gurney in the f1 frame)

iii. (1.0 pt) Which of these could fill in blank (c)?

- ◯ `3`
- ◯ `6`
- ◯ `7`
- ◯ `an arrow to list x (named gurney in the Global frame)`

iv. (1.0 pt) Which of these could fill in blank (d)?

- ◯ `3`
- ◯ `4`
- ◯ `6`
- ◯ `7`

v. (1.0 pt) Which of these could fill in blank (e)?

- ◯ `3`
- ◯ `6`
- ◯ `7`
- ◯ `an arrow to list x (named gurney in the Global frame)`
- ◯ `an arrow to list y (named gurney in the f1 frame)`

vi. (1.0 pt) Which of these could fill in blank (f)?

- ◯ `3`
- ◯ `4`
- ◯ `5`
- ◯ `6`
- ◯ `7`

vii. (1.0 pt) Which of these could fill in blank (g)?

- ◯ `an arrow to list x (named gurney in the Global frame)`
- ◯ `an arrow to list y (named gurney in the f1 frame)`
- ◯ `an arrow to list z (named gurney in the f2 frame)`
- ◯ `an arrow to another list that does not appear in the diagram`

viii. (1.0 pt) Which of these could fill in blank (h)?

- ◯ `an arrow to list x (named gurney in the Global frame)`
- ◯ `an arrow to list y (named gurney in the f1 frame)`
- ◯ `an arrow to list z (named gurney in the f2 frame)`
- ◯ `an arrow to another list that does not appear in the diagram`

(b) (6.0 points)

Implement the `Blink` class. A `Blink` instance represents a linked list of numbers and can find the longest sublist starting with any particular value in constant time. A `Blink` instance `b` is constructed from a linked list `s` (a `Link` instance or `Link.empty`) and has the following attributes:

- `b.link` is `s`, the linked list from which `b` was constructed.
- `b.rest` is a `Blink` representing the rest of `s`. If `b` represents `Link.empty`, then it has no `rest` attribute.
- `b.sublists` is a dictionary with a key for each unique element in `s`. The value for a key `k` is the `Link` instance representing the longest sublist of `s` starting with `k`.

```
class Blink:
    """A Blink has link, rest, and.sublists attributes for a linked list s.

    >>> s = Link(3, Link(1, Link(4, Link(1, Link(5)))))
    >>> b = Blink(s)
    >>> b.link is s
    True
    >>> b.rest.rest.link is s.rest.rest
    True
    >>> b.rest.rest.rest.rest.rest.link is Link.empty
    True
    >>> b.sublists[4]
    Link(4, Link(1, Link(5)))
    >>> b.sublists[1]
    Link(1, Link(4, Link(1, Link(5))))
    >>> b.rest.rest.sublists[1]
    Link(1, Link(5))
    >>> b.sublists[3] is s
    True
    """
    def __init__(self, s):
        assert s is Link.empty or isinstance(s, Link)

        if s is not Link.empty:

            self.rest = _________
                            (a)

            # Copy the sublists dict of self.rest into a new dict.
            self.sublists = self.rest.sublists.copy()

            _________
               (b)

        else:

            self.sublists = _________
                                (c)

        self.link = _________
                        (d)
```

i. (1.0 pt) Fill in blank (a).

ii. (3.0 pt) Fill in blank (b).

iii. (1.0 pt) Which of these could fill in blank (c)?

- ◯ `self.rest.sublists`
- ◯ `self.rest.sublists.copy()`
- ◯ `{}`
- ◯ `{s: s}`
- ◯ `{s: self}`
- ◯ `{s.first: s}`
- ◯ `{s.first: self}`

iv. (1.0 pt) Which of these could fill in blank (d)?

- ◯ `s`
- ◯ `s.rest`
- ◯ `s.link`
- ◯ `self`
- ◯ `self.rest`
- ◯ `self.rest.link`

2. (28.0 points) Arrakis

Definition. A *worm* is a non-negative integer in which the absolute difference between each pair of adjacent digits is 1. 4345 is a worm. 4334 and 4354 are not. All non-negative integers below 10 are worms.

You may use `near` in the problems below.

```
def near(i, j):
    """Return whether digits i and j have absolute difference equal to 1."""
    assert i >= 0 and i < 10 and j >= 0 and j < 10
    return abs(i - j) == 1
```

(a) (4.0 points)

Implement `is_worm`, which takes a non-negative integer `n` and returns `True` if `n` is a worm and `False` otherwise.

```
def is_worm(n):
    """Return whether non-negative n is a worm.

    >>> [is_worm(0), is_worm(4), is_worm(4345), is_worm(4334), is_worm(4354)]
    [True, True, True, False, False]
    >>> [n for n in range(200, 300) if is_worm(n)]
    [210, 212, 232, 234]
    """
    if _________:
          (a)

        _________
           (b)

    return near(n % 10, _________) and is_worm(_________)
                           (c)                    (d)
```

i. (1.0 pt) Which of these could fill in blank (a)?

- ◯ `n == 0`
- ◯ `n <= 0`
- ◯ `n < 10`
- ◯ `n > 0 and n < 10`

ii. (1.0 pt) Fill in blank (b).

iii. (1.0 pt) Which of these could fill in blank (c)?

- ◯ `n % 10`
- ◯ `n // 10`
- ◯ `n % 100`
- ◯ `n // 100`
- ◯ `(n % 10) // 10`
- ◯ `(n // 10) % 10`
- ◯ `(n % 10) // 100`
- ◯ `(n // 100) % 10`

iv. (1.0 pt) Which of these could fill in blank (d)?

- ◯ `n % 10`
- ◯ `n // 10`
- ◯ `n % 100`
- ◯ `n // 100`
- ◯ `(n % 10) // 10`
- ◯ `(n // 10) % 10`
- ◯ `(n % 10) // 100`
- ◯ `(n // 100) % 10`

(b) (7.0 points)

Implement `sandworm`, which takes a non-negative integer `n` and returns the largest worm that appears among the digits of `n` in order (but not necessarily formed of adjacent digits).

A worm `a` is larger than a worm `b` if `a > b`. Worms are integers.

```
def sandworm(n):
    """Return the largest worm formed by selecting some digits of non-negative n.

    >>> sandworm(13531)           # 13[5]31
    5
    >>> sandworm(152)             # [1]5[2]
    12
    >>> sandworm(31415926535)     # [3]1[4]1[5]92[6]53[5]
    34565
    >>> sandworm(314159265358973)  # [3]1[4]1[5]92[6]53589[7]3
    34567
    """
    if n == 0:
        return 0

    def use_last(n):
        "Return the largest worm in n that includes n % 10"
        return tooth(n // 10, n % 10)

    def tooth(n, d):
        "Return the largest worm formed by some digits of n followed by digit d."

        if n == 0:

            return _________
                      (a)

        skip = _________
                  (b)

        if near(n % 10, d):

            return max(skip, _________)
                                (c)

        else:

            return skip

    return max(_________, use_last(n))
                  (d)
```

i. (2.0 pt) Fill in blank (a).

ii. (2.0 pt) Fill in blank (b).

iii. (2.0 pt) Which of these could fill in blank (c)?

- ◯ `10 * use_last(n) + n % 10`
- ◯ `10 * use_last(n // 10) + n % 10`
- ◯ `10 * sandworm(n) + n % 10`
- ◯ `10 * sandworm(n // 10) + n % 10`
- ◯ `10 * use_last(n) + d`
- ◯ `10 * use_last(n // 10) + d`
- ◯ `10 * sandworm(n) + d`
- ◯ `10 * sandworm(n // 10) + d`

iv. (1.0 pt) Which of these could fill in blank (d)?

- ◯ `sandworm(n // 10)`
- ◯ `use_last(n // 10)`
- ◯ `tooth(n // 10, 0)`
- ◯ `10 * sandworm(n // 10) + n % 10`
- ◯ `10 * use_last(n // 10) + n % 10`
- ◯ `10 * tooth(n // 10, 0) + n % 10`

(c) (7.0 points)

Implement `thumper`, a generator function that takes a positive integer `k` and a digit `m`. It yields all `k`-digit worms with digits that are all less than or equal to `m`, and it yields these results in increasing order. Assume the number 0 has 0 digits.

```
def thumper(k, m):
    """Yield all k-digit worms with digits that are at most m, in increasing order.

    >>> list(thumper(1, 7))  # Note: 0 has no digits, so it is not a 1-digit worm.
    [1, 2, 3, 4, 5, 6, 7]
    >>> list(thumper(2, 3))
    [10, 12, 21, 23, 32]
    >>> list(thumper(3, 3))
    [101, 121, 123, 210, 212, 232, 321, 323]
    """
    if k == 1:

        _________
           (a)

    else:

        for w in _________:
                    (b)

            if _________:
                  (c)

                yield 10 * w + (w % 10 - 1)

            if _________:
                  (d)

                yield 10 * w + (w % 10 + 1)
```

i. (3.0 pt) Fill in blank (a).

ii. (2.0 pt) Fill in blank (b).

iii. (1.0 pt) Which of these could fill in blank (c)?

- ◯ `True`
- ◯ `False`
- ◯ `w > 0`
- ◯ `w >= 0`
- ◯ `w < m`
- ◯ `w <= m`
- ◯ `w % 10 > 0`
- ◯ `w % 10 >= 0`
- ◯ `w % 10 < m`
- ◯ `w % 10 <= m`

iv. (1.0 pt) Which of these could fill in blank (d)?

- ◯ `True`
- ◯ `False`
- ◯ `w > 0`
- ◯ `w >= 0`
- ◯ `w < m`
- ◯ `w <= m`
- ◯ `w % 10 > 0`
- ◯ `w % 10 >= 0`
- ◯ `w % 10 < m`
- ◯ `w % 10 <= m`

(d) (8.0 points)

Implement `segment`, which takes a positive integer `n` (such as `3456`) and a two-argument function `grouped`. It returns a linked list `s` containing linked lists of digits (such as `<<3 4> <5 6>>`). Together, the elements of `s` contain all digits of `n` in order. Two adjacent digits `a` and `b` (with `a` to the left of `b`) appear in the same element of `s` if `grouped(a, b)` returns a true value.

The `Link` class appears on page 2 (left column) of the midterm 2 study guide.

```
def segment(n, grouped):
    """Return a linked list of linked lists of the digits of positive n.
    Adjacent digits a and b appear in the same linked list if grouped(a, b).

    >>> print(segment(3233344, lambda a, b: a == b))
    <<3> <2> <3 3 3> <4 4>>
    >>> print(segment(314159, lambda a, b: a == 1))
    <<3> <1 4> <1 5> <9>>
    """
    part = Link.empty
    parts = Link.empty

    while n:

        if part is Link.empty or _________:
                                     (a)

            _________
               (b)

        else:

            _________
               (c)

            part = _________
                      (d)

        _________
           (e)

    return _________
              (f)
```

i. (2.0 pt) Fill in blank (a).

ii. (2.0 pt) Which of these could fill in blank (b)?

- ◯ `part = Link(n, part)`
- ◯ `part = Link(n % 10, part)`
- ◯ `part = Link(n // 10 % 10, part)`
- ◯ `part.rest = Link(n)`
- ◯ `part.rest = Link(n % 10)`
- ◯ `part.rest = Link(n // 10 % 10)`

iii. (1.0 pt) Which of these could fill in blank (c)?

- ◯ `parts.append(part)`
- ◯ `parts.rest = Link(part)`
- ◯ `parts = Link(part, parts)`
- ◯ `parts = part + parts`
- ◯ `parts += part`
- ◯ `parts += Link(part)`

iv. (1.0 pt) Which of these could fill in blank (d)?

- ◯ `part.first`
- ◯ `part.rest`
- ◯ `Link.empty`
- ◯ `Link(n % 10)`
- ◯ `Link(n, part)`
- ◯ `Link(n % 10, part)`

v. (1.0 pt) Fill in blank (e).

vi. (1.0 pt) Fill in blank (f).

(e) (2.0 points)

Implement `desert`, which takes a positive integer `n`. It returns a linked list `s` containing linked lists of digits. Together, the elements of `s` contain all digits of `n` in order, and `s` is the shortest linked list for which each element contains the digits of a worm. Assume that `segment` is implemented correctly.

```
def desert(n):
    """Return the shortest linked list whose elements are linked lists
    of digits of worms that together are the digits of positive n.

    >>> print(desert(43587))
    <<4 3> <5> <8 7>>
    >>> print(desert(11235813213455))
    <<1> <1 2 3> <5> <8> <1> <3 2 1> <3 4 5> <5>>
    """
    return _________
                (a)
```

i. (2.0 pt) Fill in blank (a).

3. (8.0 points) Caladan

Definition. A *fruit* is a **leaf node** that has a parent but no siblings. That is, its parent has no other children.

The `Tree` class appears on page 2 (left column) of the midterm 2 study guide.

You may use `fruited_branch` in the problems below.

```
def fruited_branch(t):
    """Return whether Tree t has exactly one child that is a fruit (a leaf with no siblings).

    >>> fruited_branch(Tree(4))
    False
    >>> fruited_branch(Tree(4, [Tree(5)]))
    True
    >>> fruited_branch(Tree(4, [Tree(5, [Tree(6)])]))
    False
    """
    return len(t.branches) == 1 and t.branches[0].is_leaf()
```

(a) (4.0 points)

Implement `sum_fruit_labels`, which takes a `Tree` instance `t`. It returns the sum of the labels of the fruits in `t`. If `t` has no fruits, 0 is returned.

```
def sum_fruit_labels(t):
    """Return the sum of the labels of the fruits of Tree t.

    >>> apple = Tree(5, [Tree(6, [Tree(7)]), Tree(8), Tree(9, [Tree(10)])])
    >>> sum_fruit_labels(apple)  # 7 + 10
    17
    >>> pineapple = Tree(3, [Tree(4), apple, apple, Tree(1, [Tree(2)])])
    >>> sum_fruit_labels(pineapple)  # 7 + 10 + 7 + 10 + 2
    36
    >>> sum_fruit_labels(Tree(3, [Tree(4), Tree(5)]))  # No fruits!
    0
    """
    if fruited_branch(t):
        return _________
                  (a)
    else:
        return _________
                  (b)
```

i. (2.0 pt) Fill in blank (a).

ii. (2.0 pt) Fill in blank (b).

(b) (4.0 points)

Implement `pruned`, which takes a `Tree` instance `t`. If `t` contains at least one fruit, it returns a `Tree` instance with only the nodes of `t` that appear on a path from the root to a fruit. If `t` contains no fruit, `pruned(t)` returns `None`. Calling `pruned(t)` should not modify `t`.

```
def pruned(t):
    """Return a Tree with only the nodes of t that are on a path to a fruit.

    >>> t = Tree(5, [Tree(6, [Tree(7)]), Tree(8), Tree(9, [Tree(10)])])
    >>> pruned(t)
    Tree(5, [Tree(6, [Tree(7)]), Tree(9, [Tree(10)])])
    >>> t  # t is not modified by calling pruned(t)
    Tree(5, [Tree(6, [Tree(7)]), Tree(8), Tree(9, [Tree(10)])])
    >>> pruned(Tree(2, [Tree(3), Tree(4)])) is None  # No fruit!
    True
    """
    if fruited_branch(t):

        return _________
                  (a)

    cut = [pruned(b) for b in t.branches]  # Some items in cut might be None

    if _________:
          (b)

        return _________
                  (c)
```

i. (1.0 pt) Which of these could fill in blank (a)?

◯ `t`

◯ `t.branches[0]`

◯ `Tree(t.label)`

◯ `Tree(t.label, t.branches[0])`

◯ `Tree(t.label, [b for b in t.branches if fruited_branch(b)])`

ii. (1.0 pt) Which of these could fill in blank (b)?

◯ `cut`

◯ `cut is not None`

◯ `None in cut`

◯ `any(cut)`

◯ `all(cut)`

iii. (2.0 pt) Fill in blank (c).

4. (15.0 points) Spice

Definition. A *repeated call* is a nested call expression in which each subexpression is either a number, a symbol, or a call with exactly one operand. For example, `(((f 2) 3) 4)` is a repeated call.

Reminder. In Scheme, the call expression `(f 2)` is a 2-element list containing the symbol `f` and the number 2. Therefore, one expression can evaluate to another expression. For example, the expression `(list 'f 2)` evaluates to `(f 2)`.

(a) (4.0 points)

Implement `repeated-call`, a procedure that takes an `operator` expression and a list of `operand` expressions. It returns a repeated call for the `operator` and `operands`. If `operands` is `nil`, the result is the `operator` expression.

```
;;; Construct a repeated call expression from an operator and a list of operands.
;;;
;;; scm> (repeated-call 'f '(2 3 4))
;;; (((f 2) 3) 4)
;;; scm> (repeated-call '(f 2) '(3 4))
;;; (((f 2) 3) 4)
;;; scm> (repeated-call 'f nil)
;;; f
(define (repeated-call operator operands)
    (if (null? operands)
        operator

        (_________ _________ _________)))
            (a)       (b)       (c)
```

i. (1.0 pt) Which of these could fill in blank (a)?

- ◯ `cons`
- ◯ `cdr`
- ◯ `list`
- ◯ `append`
- ◯ `map`
- ◯ `repeated-call`

ii. (2.0 pt) Fill in blank (b).

iii. (1.0 pt) Which of these could fill in blank (c)?

- ◯ `operands`
- ◯ `(cdr operands)`
- ◯ `(cons operator operands)`
- ◯ `(cons operator (cdr operands))`
- ◯ `(repeated-call operator operands)`
- ◯ `(repeated-call operator (cdr operands))`

(b) (4.0 points)

Complete the implementation of `curry`, a higher-order procedure that is called repeatedly on a non-negative integer `num-args` and then a procedure `f`. It returns a curried version of `f` that, when called repeatedly `num-args` times, returns the result of applying `f` to those arguments. Assume that `f` can take `num-args` arguments.

As a special case, `((curry 0) f)` calls `f` on no arguments, which is equivalent to evaluating `(f)`.

Hint: The built-in `apply` procedure takes a procedure `f` and a list of arguments `s` and applies `f` to the elements of `s`. For example,

- `(apply + '(1 2 3))` is equivalent to `(+ 1 2 3)` and evaluates to 6.
- `(apply + '())` is equivalent to `(+)` and evaluates to 0.

```
;;; Return a curried version of f that can be called repeatedly num-args times.
;;;
;;; scm> (((((curry 3) +) 4) 5) 6)     ;  (+ 4 5 6) evaluates to 15
;;; 15
;;; scm> ((curry 0) +)                 ; (+) evaluates to 0
;;; 0
;;; scm> (((curry 1) +) 3)             ; (+ 3) evaluates to 3
;;; 3
;;; scm> (((((curry 3) list) 4) 5) 6)  ; (list 4 5 6) evaluates to (4 5 6)
;;; (4 5 6)
(define (curry num-args)

    (lambda (f) (curry-helper num-args (lambda (s) (apply f s)))))

;;; curry-helper's argument g is a one-argument procedure that takes a list.
;;;
;;; scm> ((((curry-helper 3 cdr) 5) 6) 7)  ; (cdr '(5 6 7)) => (6 7)
;;; (6 7)
(define (curry-helper num-args g)

    (if (= num-args 0)

        _________
           (a)

        (lambda (x) (curry-helper (- num-args 1) _________))))
                                                    (b)
```

i. (2.0 pt) Fill in blank (a).

ii. (2.0 pt) Fill in blank (b).

(c) (7.0 points)

Implement `one-arg`, which takes a Scheme expression `s`. It returns a call expression that would evaluate to the same value as `s` (calling the same procedures), but which uses `curry` to ensure that all call expressions have exactly one operand. Call expressions that already have one operand are unchanged.

- Assume `s` contains only numbers, symbols, and call expressions; no special forms.
- Assume that each operator (first sub-expression) of a call expression in `s` is a symbol (such as `+`).
- Assume that each operand of a call expression in `s` is either a number or another call expression.

```
;;; Take a (possibly nested) call expression s and return
;;; an equivalent expression in which all calls have one argument.
;;;
;;; scm> (one-arg '(abs 3))  ; (abs 3) already takes just 1 argument
;;; (abs 3)
;;;
;;; scm> (+ 4 5 6)
;;; 15
;;; scm> (one-arg '(+ 4 5 6))
;;; ((((curry 3) +) 4) 5) 6)
;;; scm> (eval (one-arg '(+ 4 5 6)))  ; Same value as (+ 4 5 6)
;;; 15
;;;
;;; scm> (one-arg '(+ (- 4) (*) (* 5 6)))
;;; ((((curry 3) +) (- 4)) ((curry 0) *)) ((((curry 2) *) 5) 6))
(define (one-arg s)

    (if (number? s) s

        (let ((num-args (- (length s) 1)))

            (if (= num-args 1)

                (_________ _________ (one-arg _________)))
                    (a)       (b)                 (c)

                (repeated-call (list _________ _________)
                                        (d)       (e)

                               (map _________ (cdr s)))))))
                                       (f)
```

i. (1.0 pt) Which of these could fill in blank (a)?

◯ `cons`

◯ `car`

◯ `cdr`

◯ `list`

◯ `append`

◯ `length`

ii. (1.0 pt) Which of these could fill in blank (b)?

- ◯ `s`
- ◯ `(car s)`
- ◯ `(cdr s)`
- ◯ `(car (cdr s))`
- ◯ `(car (cdr (cdr s)))`

iii. (1.0 pt) Which of these could fill in blank (c)?

- ◯ `s`
- ◯ `(car s)`
- ◯ `(cdr s)`
- ◯ `(car (cdr s))`
- ◯ `(car (cdr (cdr s)))`

iv. (2.0 pt) Fill in blank (d).

v. (1.0 pt) Which of these could fill in blank (e)?

- ◯ `s`
- ◯ `'s`
- ◯ `(car s)`
- ◯ `'(car s)`
- ◯ `(cdr s)`
- ◯ `'(cdr s)`

vi. (1.0 pt) Which of these could fill in blank (f)?

- ◯ `one-arg`
- ◯ `car`
- ◯ `cdr`
- ◯ `(lambda (x) (car (cdr x)))`
- ◯ `(lambda (x) (one-arg (car x)))`
- ◯ `(lambda (x) (one-arg (car (cdr x))))`

5. (10.0 points) Gom Jabbar

(a) (4.0 pt) Which of the following strings is entirely matched (from beginning to end) by the regular expression below? Check all that apply.

`(([cs][61]|[abc])?(cs)+)|([cs]+61+)`

- ☐ `cs61a`
- ☐ `cs61acscs`
- ☐ `cs61`
- ☐ `cscs`
- ☐ `cscs611`
- ☐ `cccsss61`
- ☐ `cs6161`
- ☐ `s1cs`

(b) (2.0 pt) Write a short string (fewer than 10 characters) that matches the BNF grammar below, but is guaranteed to cause an error when evaluated by Scheme, regardless of how any symbols such as `f` are defined in the environment.

Notes: The `%ignore /\s+/` directive ignores whitespace in the string. The `INT` terminal matches integers.

```
?start: expr
expr: INT | "(" operator expr+ ")"
operator: PROCEDURE | expr
PROCEDURE: "f"

%ignore /\s+/
%import common.INT
```

(c) (4.0 pt) Write a SQL query that generates a one-column table of the names of all dogs with exactly one child that has short fur. (The dog may have multiple children, but only one can have short fur.)

Assume the `parents` and `dogs` tables from page 1 (right column) of the final study guide have been created.

The result of your query should contain two rows: "Abraham" and "Fillmore". Your query should select the rows described even if the contents of `parents` and `dogs` were different; no credit for `SELECT "Abraham" UNION SELECT "Fillmore"`.

6. (0.0 points) Just for Fun

Draw a picture of something you enjoyed about CS 61A.

No more questions.