CS 61A Fall 2023

Structure and Interpretation of Computer Programs

Final Solutions

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?

(c) What is your @berkeley.edu email address?

(d) Sign (or type) your name to confirm that all work on this exam will be your own. The penalty for academic misconduct on an exam is an F in the course.

1. (5.0 points) Copying Copies

Draw the environment diagram that results from running all of the code below until it is fully executed, an error occurs, or you run out of frames. Then, answer the questions that follow. Blanks and frames with no labels have no questions associated with them and are not scored. You may not need all the spaces or frames.

m43qkom.png

(a) (1.0 pt) Which of the following is true about the blanks labeled (a) and (b)?

- ● They contain arrows to the same list
- ○ They contain arrows to different lists with the same contents
- ○ They contain arrows to different lists with different contents

(b) (1.0 pt) What error occurs during execution, if any?

- ○ `NameError` because a name is referenced before assignment
- ○ `TypeError` because a function was called with the wrong number of arguments
- ○ `TypeError` because a built-in function is called on the wrong argument type(s)
- ○ `RecursionError` because of too many recursive calls
- ● No error occurs

(c) (3.0 pt) What would be displayed by evaluating `print(result)` in the global frame? If an error occurred or you ran out of frames, still evaluate `print(result)` according to the environment diagram you drew.

```
[5, 1, [5, 1], 2]
```

2. (6.0 points) Path Math

Implement `bounds`, which takes a `Tree` instance `t` and numbers `low` and `high`. It returns the number of paths through `t` from the root to a leaf for which the sum of the labels along the path is at least `low` and at most `high`.

```
def bounds(t, low, high):
    """Return the number of root-to-leaf paths in t whose sum is between low and high (inclusive).

    >>> t = Tree(3, [Tree(4), Tree(5, [Tree(1), Tree(2)]), Tree(7)])
    >>> bounds(t, 7, 10)  # 3+4=7, 3+5+1=9, 3+5+2=10, 3+7=10
    4
    >>> bounds(t, 9, 10)
    3
    >>> bounds(t, 9, 9)
    1
    """
    count = 0

    if _______:
         (a)

        count = 1

    return count + _______([_______ for b in t.branches])
                     (b)      (c)
```

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

- ○ `low <= t and t <= high`
- ○ `t and low <= t and t <= high`
- ○ `t.branches and low <= t and t <= high`
- ○ `t.is_leaf() and low <= t and t <= high`
- ○ `low <= t.label and t.label <= high`
- ○ `t and low <= t.label and t.label <= high`
- ○ `t.branches and low <= t.label and t.label <= high`
- ● `t.is_leaf() and low <= t.label and t.label <= high`

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

- ○ `bounds`
- ○ `max`
- ● `sum`
- ○ `lambda t:`

(c) **(3.0 pt)** Fill in blank (c).

```
bounds(b, low - t.label, high - t.label)
```

3. (10.0 points) Talk Like a Pirate Day

Pirate expressions are reversed and substitute some words (such as `"aye"` for `"yes"`) according to a pirate dialect.

- An `Expression` instance is constructed from a list of strings and has a dictionary attribute `dialect` and a `Word` instance attribute `first` that represents the first word of the reversed sequence.
- A `Word` instance is constructed from its attributes: a string `w`, an `Expression` instance `exp`, and a `Word` instance `then` representing the next word in the reversed sequence. If there is no next word, `then` is `None`. A `Word` instance's `say` method returns either `w` or its substitute if `w` is a key of the expression's `dialect`.

Printing a `Word` prints how a pirate would `say` that word and the following words in the reversed sequence. Printing an `Expression` prints all of the words in the reversed sequence using the `Expression`'s dialect.

Reminder: The `get` method of a dictionary takes two arguments: `key` and `default`. If the `key` is in the dictionary, its value is returned. If not, `default` is returned. E.g., `{1:2}.get(1, 3)` evaluates to 2, but `{1:2}.get(5, 3)` is 3.

```
class Expression:
    """A pirate expression is reversed and substitutes some words using a dialect.
    >>> str(Expression(['I', 'said', 'hi']))
    'ahoy says I'
    >>> e = Expression(['there', 'you', 'are'])
    >>> print(e)
    arrrr you there
    >>> e.dialect['you'] = 'ye'  # After adding to the dialect...
    >>> print(e)                 # ... the result of printing changes
    arrrr ye there
    """
    def __init__(self, original):
        assert len(original) > 0
        self.dialect = {'yes': 'aye', 'hi': 'ahoy', 'said': 'says', 'are': 'arrrr'}
        previous = None
        for w in original:
            current = _______
                        (a)

            _______
              (b)
        self.first = _______
                       (c)
    def __str__(self):
        return _______
                 (d)
class Word:
    def __init__(self, w, exp, then):
        self.w = w
        self.exp = exp
        self.then = then

    def say(self):
        return _______
                 (e)
    def __str__(self):
        first = _______
                  (f)
        if self.then:
            return first + ' ' + str(self.then)
```

```
    else:
        return first
```

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

```
Word(w, self, previous)
```

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

```
previous = current
```

(c) **(1.0 pt)** Fill in blank (c).

- ● `current`
- ○ `self.current`
- ○ `current.then`
- ○ `self.current.then`

(d) **(1.0 pt)** Fill in blank (d).

- ○ `self`
- ○ `self.first`
- ○ `str(self)`
- ● `str(self.first)`
- ○ `print(self)`
- ○ `print(self.first)`

(e) **(3.0 pt)** Fill in blank (e).

```
self.exp.dialect.get(self.w, self.w)
```

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

- ○ `self.w`
- ○ `self.exp[w]`
- ○ `self.say`
- ● `self.say()`
- ○ `self.say(self)`
- ○ `self.say(self.w)`

4. (29.0 points) Six Pages of Pairings (So Please Read the Definition Carefully)

Definition: A *pairing* of a sequence `s` is a list of two-element tuples (called *pairs*) that contain adjacent elements of `s`. There must be at least one element of `s` **between** any two pairs in the pairing. The pairs in the pairing must be in the same order as they are in `s`.

The sequence `[4, 5, 6, 1, 2, 3, 7, 8]` has pairings `[(6, 1), (3, 7)]` and `[(4, 5), (1, 2), (7, 8)]` and `[]` and many others, but the following are not pairings of that sequence:

- `[(5, 1), (3, 7)]` contains a pair `(5, 1)`, but 5 and 1 are not adjacent in the sequence.
- `[(5, 6), (1, 2)]` contains two pairs with no element between them, since 6 is adjacent to 1 in the sequence.
- `[(1, 2), (4, 5)]` contains valid pairs, but those pairs are not in the same order as the sequence.

(a) (6.0 points)

Implement `longest_pairing`, which takes a list `s` with `3*n-1` elements for some positive integer `n`. It returns the longest pairing of `s`.

```
def longest_pairing(s):
    """Return the longest pairing for list s that has length 3*n-1 for some positive integer n.

    >>> longest_pairing([1, 2, 3, 4, 5, 6, 7, 8])
    [(1, 2), (4, 5), (7, 8)]
    """
    assert len(s) > 0 and _______, 's must have length 3*n-1 for a positive integer n'
                            (a)

    result, pair, skip = [], [], False

    for x in s:

        if _______:
            (b)

            pair.append(x)

        else:

            _______
              (c)

        if _______:
            (d)

            results._______
                      (e)

            pair, skip = [], True

    return result
```

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

- ○ `len(s) == 3 * n - 1`
- ○ `(len(s) - 1) / 3`
- ○ `(len(s) - 1) % 3 == 0`
- ● `len(s) % 3 == 2`

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

- ○ `skip`
- ● `not skip`
- ○ `result`
- ○ `not result`
- ○ `pair`
- ○ `not pair`
- ○ `len(pair) < 2`
- ○ `len(pair) == 2`

iii. (1.0 pt) Fill in blank (c)

- ● `skip = False`
- ○ `skip = True`
- ○ `pair = []`
- ○ `pair.remove(x)`
- ○ `result.append(x)`
- ○ `result.append(pair)`

iv. (1.0 pt) Fill in blank (d)

- ○ `skip`
- ○ `not skip`
- ○ `result`
- ○ `not result`
- ○ `pair`
- ○ `not pair`
- ○ `len(pair) < 2`
- ● `len(pair) == 2`

v. (2.0 pt) Fill in blank (e). **Select all that apply.**

- ☐ `append(pair)`
- ☐ `extend(pair)`
- ■ `append((pair[0], pair[1]))`
- ☐ `extend((pair[0], pair[1]))`
- ■ `append(tuple(pair))`
- ☐ `extend(tuple(pair))`
- ☐ `append(tuple(pair[0], pair[1]))`
- ☐ `extend(tuple(pair[0], pair[1]))`

(b) (4.0 points)

Implement `is_pair_sequence`, which takes a list `s` and returns whether it contains only two-element tuples.

```
def is_pair_sequence(s):
    """Return whether list s contains only pairs (which are tuples with two elements).

    >>> is_pair_sequence([(1, 2), (3, 4)])
    True
    >>> is_pair_sequence([(1, 2), (3, 4, 5)])
    False
    >>> is_pair_sequence([(1, 2), "not a tuple"])
    False
    >>> is_pair_sequence([(1, 2), (3, (4, 5, 6))])  # (3, (4, 5, 6)) is a two-element tuple
    True
    >>> is_pair_sequence([])
    True
    """
    return all([_______ for x in s]) and all(map(_______, s))
                  (f)                                (g)
```

i. (2.0 pt) Fill in blank (f). **Select all that apply.** Assume `tuple` has no subclasses.

- [ ] `tuple(x)`
- [ ] `x == tuple`
- [ ] `x is tuple`
- [x] `type(x) == tuple`
- [ ] `x == type(tuple)`
- [x] `isinstance(x, tuple)`
- [ ] `isinstance(x, type(tuple))`
- [ ] `isinstance(type(x), tuple)`

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

- ○ `len(s) == 2`
- ○ `len(x) == 2`
- ○ `lambda x: len(s) == 2`
- ● `lambda x: len(x) == 2`
- ○ `lambda s: lambda x: len(s) == 2`
- ○ `lambda s: lambda x: len(x) == 2`
- ○ `lambda x: len(s[i]) == 2`
- ○ `lambda x: len(x[i]) == 2`
- ○ `lambda i: lambda x: len(s[i]) == 2`
- ○ `lambda i: lambda x: len(x[i]) == 2`

(c) (6.0 points)

Implement `is_pairing`, which takes a list `s` and a list of `pairs`. It returns whether `pairs` is a pairing of `s`.

```
def is_pairing(s, pairs):
    """Return whether the list of pairs is a pairing for the list s.
    >>> pairs = [(3, 4), (5, 6), (7, 7)]
    >>> is_pairing([3, 3, 4, 5, 4, 5, 6, 0, 7, 7, 7], pairs)
    True
    >>> is_pairing([3, 3, 4, 5, 6, 0, 7, 7, 7], pairs)  # Need an element between pairs
    False
    >>> is_pairing([3, 2, 4, 0, 5, 6, 0, 7, 7], pairs)  # Elements of a pair must be adjacent
    False
    >>> is_pairing([7, 7, 3, 3, 4, 5, 4, 5, 6], pairs)  # Pairing isn't in the same order as s
    False
    """
    assert is_pair_sequence(pairs)
    if not pairs:
        return True
    if _______:
         (h)
        return False
    if _______ == tuple(s[:2]):
         (i)
        return is_pairing(s[3:], _______)  # Note: [0, 1][3:] evaluates to []
                                   (j)
    return _______
            (k)
```

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

- ○ `pairs not in s`
- ○ `pairs[0] not in s`
- ● `len(s) < 2`
- ○ `not is_pairing(s, pairs)`

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

`pairs[0]`

iii. (1.0 pt) Fill in blank (j).

- ○ `pairs`
- ○ `pairs[1]`
- ● `pairs[1:]`
- ○ `pairs[:1]`

iv. (3.0 pt) Fill in blank (k).

```
is_pairing(s[1:], pairs)
```

(d) (7.0 points)

Implement `unequal_pairs`, a generator function that yields all **non-empty** pairings of a list `s` in which no pair contains two equal elements.

```
def unequal_pairs(s):
    """Yield all non-empty pairings for a list s in which each pair's values are unequal.

    >>> sorted(unequal_pairs([4, 2, 2, 4, 4, 1, 1]))  # Four different pairings!
    [[(2, 4)], [(4, 1)], [(4, 2)], [(4, 2), (4, 1)]]
    >>> max(unequal_pairs([4, 2, 2, 4, 5, 4, 4, 1, 5, 5, 6]), key=len)  # The longest pairing
    [(4, 2), (4, 5), (4, 1), (5, 6)]
    """
    if len(s) >= 2:

        yield from _______
                     (l)

        if _______:
             (m)

            pair = (s[0], s[1])

            _______
              (n)

            for rest in unequal_pairs(s[3:]):  # Note: [0, 1][3:] evaluates to []

                yield _______
                         (o)
```

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

`unequal_pairs(s[1:])`

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

- ○ `s[0] == s[1]`
- ● `s[0] != s[1]`
- ○ `pair[0] == pair[1]`
- ○ `pair[0] != pair[1]`

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

`yield [pair]`

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

`[pair] + rest`

(e) (6.0 points)

Implement `max_pair_sum`, which takes a linked list `s` (either a `Link` instance or `Link.empty`). It returns the largest possible sum of the values in a pairing of `s`. The `Link` class appears on the Midterm 2 Study Guide (p. 2).

```
def max_pair_sum(s):
    """Return the largest sum of values in a pairing for a linked list of positive numbers s.

    >>> L = Link                                                        # Abbreviate Link
    >>> max_pair_sum(L(3, L(4, L(5, L(3, L(4, L(5, L(6))))))))         # 4+5 + 5+6
    20
    >>> max_pair_sum(L(3, L(4, L(5, L(3, L(4, L(5, L(6, L(3))))))))) # 3+4 + 3+4 + 6+3
    23
    """
    if _______:
         (p)
        return 0
    n = _______
          (q)
    if s.rest is Link.empty:
        return n
    else:
        return max(n + max_pair_sum(_______), max_pair_sum(_______))
                                      (r)                     (s)
```

i. (2.0 pt) Fill in blank (p). **Select all that apply.**

- ☐ `s is Link.empty`
- ☐ `s.rest is Link.empty`
- ■ `s is Link.empty or s.rest is Link.empty`
- ☐ `s.rest is Link.empty or s is Link.empty`

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

```
s.first + s.rest.first
```

iii. (1.0 pt) Fill in blank (r).

- ○ `s`
- ○ `s.rest`
- ○ `s.rest.rest`
- ● `s.rest.rest.rest`

iv. (1.0 pt) Fill in blank (s).

- ○ `s`
- ● `s.rest`
- ○ `s.rest.rest`
- ○ `s.rest.rest.rest`

5. (6.0 points) What Would Scheme Do?

Assume the following code has been evaluated.

```
(define (shrink k t)
  (lambda (s)
    (if (null? s) t
        ( (if  (= (car s) k)  (shrink (+ k 1) t)  (shrink k (cons (car s) t)))
          (cdr s)) )))

(define-macro (wait expr) `(lambda () ,expr))
(define (double wait-list)
  (if (null? wait-list) nil
      (cons (* 2 (car wait-list)) (wait (double ((cdr wait-list)))))))

(define twos (cons 2 (wait (double twos))))
```

(a) (2.0 pt) What does this expression evaluate to? `((shrink 3 nil) '(3 1 4 1 5 9 2 6))`

`(2 9 1 1)`

(b) (1.0 pt) What is the order of growth of the run time of `((shrink 1 nil) s)` in terms of the length of list `s`?

- ○ `constant`
- ● `linear`
- ○ `quadratic`
- ○ `exponential`

(c) (1.0 pt) Which of the following evaluates to 4?

- ○ `(cdr twos)`
- ○ `((cdr twos))`
- ○ `(car (cdr twos))`
- ○ `((car (cdr twos)))`
- ● `(car ((cdr twos)))`
- ○ `((car ((cdr twos))))`

(d) (2.0 pt) Which of the following evaluates to 8?

- ○ `(cdr (cdr twos))`
- ○ `((cdr (cdr twos)))`
- ○ `(cdr ((cdr twos)))`
- ○ `(car (cdr (cdr twos)))`
- ○ `((car (cdr (cdr twos))))`
- ○ `(car (cdr ((cdr twos))))`
- ○ `(car ((cdr (cdr twos))))`
- ● `(car ((cdr ((cdr twos)))))`

6. (5.0 points) How to Get Promoted

Implement `promote`, which takes a one-argument procedure `f` and a list `s`. It returns a list that begins with all of the elements of `s` for which calling `f` on the element returns `#t`, followed by all of the elements of `s` for which calling `f` on the element returns `#f`. Assume that when `f` is called on any element of `s`, it returns either `#t` or `#f`.

```
;;; Return a list containing all the elements of s, with the elements for which
;;; f returns #t at the front of the list, but otherwise keeping the order the same.
;;;
;;; scm> (promote even? '(1 2 3 4 5 6 7))
;;; (2 4 6 1 3 5 7)
;;; scm> (promote odd?  '(1 2 3 4 5 6 7))
;;; (1 3 5 7 2 4 6)
(define (promote f s)
    (_______ _______ (filter _______ s)))
       (a)     (b)             (c)
```

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

- ● `append`
- ○ `cons`
- ○ `list`
- ○ `promote`
- ○ `map`
- ○ `filter`

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

```
(filter f s)
```

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

```
(lambda (x) (not (f x)))
```

(d) **This is an A+ question.** It is not worth any points. It is **not** the last question on the exam.

Implement `bigger-first` **without writing** `lambda` **or** `define`. You may use `promote`, `curry`, and `curry-call`.

```
(define (curry f) (lambda (x) (lambda (y) (f x y))))
(define (curry-call f) (lambda (x g) (lambda (y) (f (x (g y)) y))))

;;; (bigger-first s) returns a list with all of the elements of s larger than (car s)
;;; at the front, followed by all of the elements of s smaller or equal to (car s).
;;;
;;; scm> (bigger-first '(3 1 4 1 5 9 2 6))
;;; (4 5 9 6 3 1 1 2)
(define bigger-first _______)
```

```
((curry-call promote) (curry <) car)
```

7. (8.0 points) Don't Skip This

Definition. A *skip-partition* of a positive integer `n` is a list of positive integers in increasing order that sums to `n` and does not contain any duplicates or consecutive numbers.

Implement `part`, which takes positive integers `n` and `m`. It returns a list of all *skip-partitions* of `n` that contain elements greater than or equal to `m`. The provided `cons-me` procedure is used in the implementation.

```
(define (cons-me first) (lambda (rest) (cons first rest)))

;;; Return a list of all skip-partitions of n with elements greater than or equal to m.
;;;
;;; scm> (part 12 2)
;;; ((2 4 6) (2 10) (3 9) (4 8) (5 7) (12))
(define (part n m)
    (cond ((= m n) _______)
                     (a)
          ((> m n) nil)
          (else (append
                  (_______ (cons-me m) _______)
                     (b)                  (c)
                  (part n (+ m 1))))))
```

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

`(list (list m))`

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

○ `cons`

○ `list`

○ `append`

● `map`

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

`(part (- n m) (+ m 2))`

(d) (3.0 pt) Which expressions are passed to `scheme_eval` when evaluating `(if (> 1 2) (+ 1 2) 2)`? **Check all that apply.**

☐ `if`

■ `(> 1 2)`

■ `>`

■ `1`

■ `2`

☐ `(+ 1 2)`

☐ `+`

8. (6.0 points) Cheap Donuts

There are three tables in a database:

- The `donuts` table has a row for each menu option at a donut shop. There are columns for the `kind` (string) of dough and `flavor` (string). For example, there is one row for chocolate cake donuts (although the store may have many such donuts, they only have one menu option for this kind & flavor combination).
- The price of a donut depends only on the dough. The `prices` table has a row for each kind of dough. The `dough` (string) column contains the kind; the `price` (number) column is for one donut made from that kind of dough.
- Your friends only care about the flavor, not the kind of dough. The `quantity` table contains one row for every flavor your friends want. The `choice` (string) column is the flavor they want and the `k` (number) column is the number of donuts of that flavor they want.

Create a table with two columns, `flavor` (string) and `total` (number) with one row for each flavor your friends want. The `total` column contains the **least expensive** total cost of buying `k` donuts of that flavor, where `k` is the number your friends want.

The rows of the result can appear in any order. Here is an example, but complete the query so that it would work even if the contents or number of rows were different.

uG153m6.png

```
SELECT flavor, _______ AS total FROM _______ WHERE _______ GROUP BY _______;
                 (a)                   (b)           (c)              (d)
```

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

```
MIN(price) * k
```

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

- ○ `quantity, prices`
- ● `quantity, donuts, prices`
- ○ `donuts, prices AS a, prices AS b`
- ○ `quantity, prices AS a, prices AS b`

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

```
kind=dough AND choice=flavor
```

(d) (1.0 pt) Fill in blank (d).

- ○ `quantity`
- ○ `dough`
- ○ `kind`
- ● `flavor`

9. A+ Questions

These are two separate A+ questions. They can only affect your course grade if you have a high A and might receive an A+. Finish the rest of the exam first! There is a third A+ question earlier in the exam on Page 13.

(a) Complete the definition of `fib` so that `(prefix fib 10)` is a list of the first 10 Fibonacci numbers: `(0 1 1 2 3 5 8 13 21 34)`. A Fibonacci number is the sum of the previous two. You **may not** write `lambda` or `let` or `define`.

```
(define-macro (wait expr) `(lambda () ,expr))
(define (prefix s k) (if (zero? k) nil (cons (car s) (prefix ((cdr s)) (- k 1)))))
(define (add s t) (cons (+ (car s) (car t)) (wait (add ((cdr s)) ((cdr t))))))

(define fib (cons 0 (wait (cons 1 _______))))
```

```
(wait (add fib ((cdr fib))))
```

(b) This question uses the definitions and functions from the earlier question called *Six Pages of Pairings*.

Fill in blank (a) of `match`, a generator function that takes a list `s` and a pairing `pairs`. It yields all non-empty pairings of `s` that contain the pairs in `pairs` in order, but which may also contain other pairs as well.

Just fill in just blank (a) as your answer. The remaining blanks are not scored.

```
def match(s, pairs):
    """Yield all non-empty pairings of s that contain pairs in order.

    >>> for p in sorted(match(range(14), [(3, 4), (8, 9)])):
    ...     print(p)
    [(0, 1), (3, 4), (8, 9)]
    [(0, 1), (3, 4), (8, 9), (11, 12)]
    [(0, 1), (3, 4), (8, 9), (12, 13)]
    [(3, 4), (8, 9)]
    [(3, 4), (8, 9), (11, 12)]
    [(3, 4), (8, 9), (12, 13)]
    """
    assert is_pair_sequence(pairs)
    if len(s) >= 2:
        first = tuple(s[:2])
        if _______:  # Just fill in this blank as your answer
             (a)

            yield [first]
        if _______:
            rest = pairs[1:]
        else:
            rest = pairs
        for p in match(_______, rest):
            yield _______ + p
        yield from match(_______, pairs)
```

```
(len(pairs) == 1 and pairs[0] == first) or not pairs
```

No more questions.