UNIVERSITY OF TORONTO
Faculty of Arts and Science

AUGUST 2019 EXAMINATIONS CSC148H1Y:
Introduction to Computer Science

Duration: 3 hours Instructor(s): Misha Schwartz. **No Aids Allowed**

Name:

Student Number:

Please read the following guidelines carefully.

- Please print your name and student number on the front of the exam.
- This examination has **9** questions. There are a total of **15 pages, DOUBLE-SIDED**.
- **DO NOT** open or turn over the exam paper until the exam has started.
- You may always write helper functions unless asked not to.
- Documentation is *not* required unless asked for.
- Answer questions clearly and completely. Provide justification unless explicitly asked not to.
- **You must earn a grade of at least 40% on this exam to pass this course**.

As a student, you help create a fair and inclusive writing environment. If you possess an unauthorized aid during an exam, you may be charged with an academic offence. Turn off and place all cell phones, smart watches, electronic devices, and unauthorized study materials in your bag under your desk. If it is left in your pocket, it may be an academic offence. Students must hand in all examination materials at the end.

Take a deep breath.

This is your chance to show us

How much you've learned.

We **WANT** to give you the credit

That you've earned.

A number does not define you.

It's been a real pleasure teaching you this term.

Good luck!

Question	Grade	Out of
Q1		9
Q2		6
Q3		5
Q4		6
Q5		9
Q6		8
Q7		8
Q8		11
Q9		8
Total		70

1. **[9 marks] Short Answer Questions**.

   Each of the questions in this section can be answered in point form, with a few lines of code, or with a few sentences. You do not need to write a lot to get full marks.

   (a) **[1 mark]** Which of the following abstract data types (ADT) are an ordered collection of values that may or may not be unique. Circle all that apply:

   - Stack
   - List
   - Set
   - Multiset
   - Queue
   - Map

   (b) **[2 marks]** Briefly explain how *composition* can be used to implement a `Tree` class (see the provided aid sheet).

   (c) **[2 marks]** In general, accessing the private attributes and methods of an instance of another class is discouraged. However, accessing the private attributes and methods of an instance of the same class is not discouraged. Briefly explain why.

   (d) **[1 mark]** The big oh (worst-case) run-time of mergesort is much less than the big oh run-time of quicksort. Give one reason why quicksort is still faster in most cases.

   (e) **[2 marks]** Briefly explain the difference between a non-expression statement and an expression statement and give an example of each.

   expression: non-expression statement:

   explain the difference:

   (f) **[1 mark]** Imagine we have defined some non-abstract class `MyUsefulClass`. If we define another class as:

   ```
   class MyOtherClass(MyUsefulClass):
     pass
   ```

   Does it make a difference if we use `MyUsefulClass` or `MyOtherClass` in our code? Yes No
   Briefly explain why or why not.

2. **[6 marks] Abstract Data Types**.

Consider the partial implementation of the `Queue` abstract data type below. This implementation uses a Python dictionary to store the data in the `Queue`. For the following questions, you must use this implementation of the `Queue` class as it is given. Do not redefine any attributes or methods that have been provided.

```
class Queue:
    _data: Dict[int, Any]

    def __init__(self) -> None:
        """ Initialize an empty Queue """
        self._data = {}

    def to_multiset(self) -> List[Any]:
        """ Return a copy of the values in the Queue in an arbitrary order """
        return list(self._data.values())

    def is_empty(self) -> bool:
        """ Return True if there are no values in the Queue """
        return len(self._data) == 0
```

(a) **[2 marks]** Based on this partial implementation, implement the `Queue.enqueue` method. The big oh (worst-case) run-time of this method should be constant relative to the size of the Queue.

```
def enqueue(self, item: Any) -> None:
    """ Enqueue <item> """
```

(b) **[4 marks]**

Based on this partial implementation and your `Queue.enqueue` method, implement the `Queue.dequeue` method. The big oh (worst-case) run-time of this method should be linear relative to the size of the Queue. Remember, dictionary look-up, insertion, and deletion all run in constant time.

```
def dequeue(self) -> Any:
    """ Dequeue the next item """
```

3. **[5 marks] Linked Lists**.

   (a) **[5 marks]** Implement the following `LinkedList` method. Do not use any `LinkedList` methods in your answer (this includes the initializer). Your code should interact with the nodes directly. Your answer must make use of the template provided.

```
def keep_biggest(self, other: LinkedList[int]) -> None:
    """Remove each item in <self> that has a value less than the corresponding item in <other>.

    If the value of the item at index i in <self> is less than the value of the item at index
    i in <other>, remove the item at index i from <self>. If there is no corresponding element
    at index i in <other>, do not remove the item.

    === Precondition ===
    All items in this linked list are numbers

    >>> lst1 = LinkedList([2, 5, 10, 12, 0])
    >>> lst2 = LinkedList([1, 20, 10, 99])
    >>> str(lst1)
    '[2 -> 5 -> 10 -> 12 -> 0]'
    >>> lst1.keep_biggest(lst2)
    >>> str(lst1)
    '[2 -> 10 -> 0]'

    """
    curr1 = self._first
    curr2 = other._first









    while









        curr1 =
        curr2 =
```

4. **[6 marks] Recursion**.

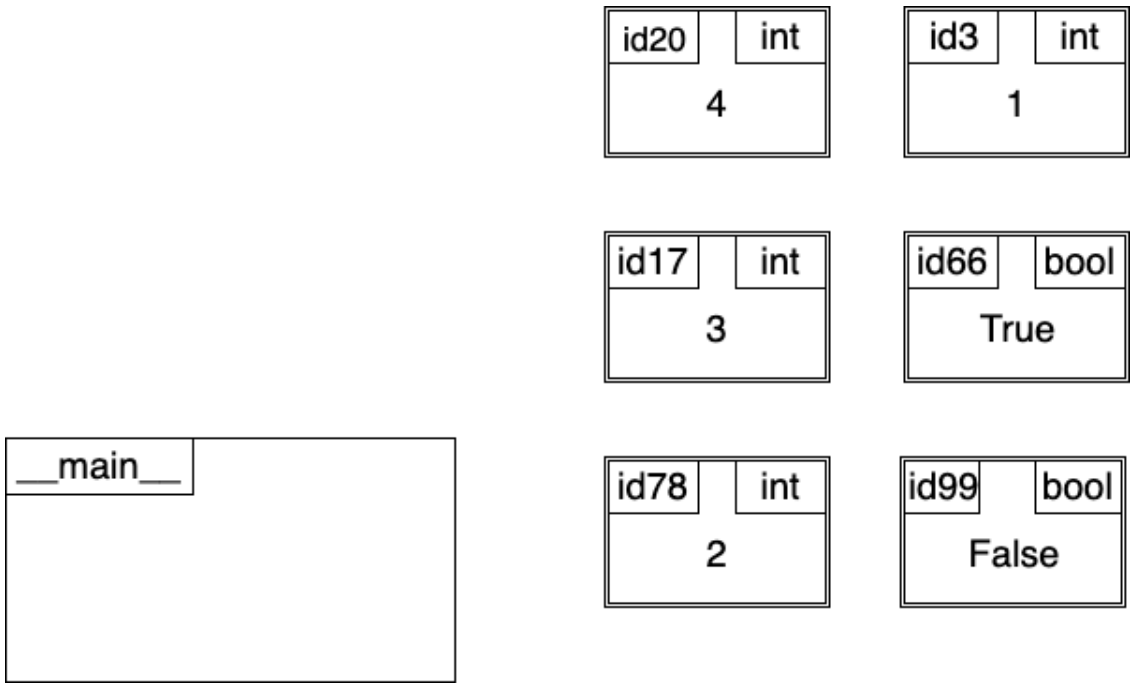
   (a) **[5 marks]** Consider the following Python function and statements:

```
def gte(x: int, y: int) -> bool:
    """ Return True if x is greater than or equal to y """
    if x == y:
        return True # draw the memory model just before this line is executed
    else:
        return gte(x - 1, y)
    return False

if __name__ == '__main__':
    x = 4
    y = 2
    result = gte(x, y)
```

   Complete the memory model below so that it shows the state of call stack just before the return statement in the base case is executed if we run the above code.

   All of the objects that you will require are already drawn for you. Do not add new objects. You are only required to fill in the call stack. Add as many additional frames to the call stack that you require.



   (b) **[1 mark]** Describe in one sentence how the call stack you drew above would be different if we replaced the final line of the code with:

```
result = gte(y, x)
```

5. **[9 marks] Fast Linked Lists.**

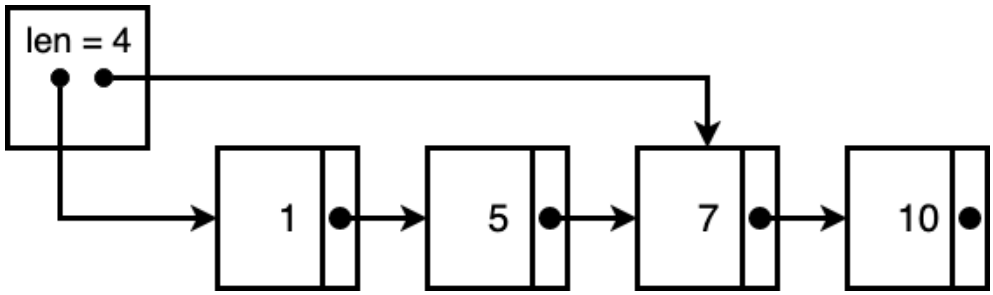
The `LinkedList` class contains a single attribute that refers to the first node in the linked list or `None` if the `LinkedList` instance is empty. A variation of this class is called a `FastLinkedList` which contains two additional attributes: one that refers to the middle node in the list, and another that stores the length of the list. A partial representation of this class is given below:

```
class FastLinkedList:
    _first: Optional[_Node]
    _middle: Optional[_Node]
    _len: int

    def __init__(self, items: Any) -> None:
        """ Initialize a FastLinkedList instance that contains <items>

        After initializing:
            self._first should refer to the node at index 0
            self._middle should refer to the node at index len(items)//2
            self._len should equal len(items)
        """
```

The middle node is the node at the index that is equal to half of the length of the list rounded down. A memory model of the `FastLinkedList` that contains the values `[1, 5, 7, 10]` looks like this:



(a) **[4 marks]** Given a `LinkedList` instance `linky` and a `FastLinkedList` instance `flinky` that both contain the same number of items, state whether the following statements are true or false and give a brief (one sentence) explanation why:

Finding the value at index `i` (where `0 <= i < len(linky)`) is always faster for `flinky` than it is for `linky`.
true false

Appending a value to the end of `flinky` is always faster than appending value to the end of `linky`.
true false

Finding some value in `flinky` is always faster than finding some value in `linky`.
true false

Inserting a value at the beginning `flinky` is always faster than inserting a value at the beginning of `linky`.
true false

(b) **[5 marks]** Implement the `FastLinkedList.insert` method below. Make sure that all the attributes refer to the correct values after the method is run. Do not use any `FastLinkedList` methods in your answer (this includes the initializer). Your code should interact with the nodes directly. **Make sure to visit as few nodes as possible while inserting**.

```
def insert(self, index: int, object: Any) -> None:
  """ Insert <object> into self so that its index is equal to <index>

  === Precondition ===
  index <= len(self)

  >>> flinky = FastLinkedList([4, 5, 6])
  >>> flinky.insert(1, 7)
  >>> str(flinky)  # you can assume FastLinkedList.__str__ is defined
  [4 -> 7 -> 5 -> 6 ]
  """
```

6. **[8 marks] Binary Search Trees.**

   (a) **[2 marks]** Assume that the `BinarySearchTree.insert` method correctly inserts a single value into an instance of a binary search tree. The resulting binary search tree will be well formed but may not necessarily be balanced. In what order should you insert the integers from 1 to 7 (inclusive) to ensure that the resulting binary search tree is balanced.

   (b) **[1 mark]** `BinarySearchTree.get_less_than` is a method that takes a single argument `i` and returns a list of all values in the binary search tree that are less than `i`. What is the best case run-time of this method in terms of `n` (n = the size of the tree)?

   Briefly explain why.

   (c) **[5 marks]** `BinarySearchTree.invert` is a method that multiplies every value stored in a `BinarySearchTree` instance by `-1`. Implement this method below. Make sure that all representation invariants for the `BinarySearchTree` still hold after this method has been run. You may access all `BinarySearchTree` attributes, but you may not use any other `BinarySearchTree` methods.

```
def invert(self) -> None:
  """Multiply every value in <self> by -1

  >>> isinstance(bst, BinarySearchTree)
  True
  >>> bst.content() # this method returns all the values in the tree in some order
  [5, -3, 2, 99]
  >>> bst.invert()
  >>> bst.content()
  [-5, 3, -2, -99]
  """
```

7. **[8 marks] Expression Trees**.

   (a) **[8 marks]** Review the Abstract Syntax Trees section of the provided aid sheet before answering this question.

   The `BinOp.evaluate` method may raise a `TypeError` depending on the type of expression in the `left` and `right` attributes. Remember, that in Python you can sometimes add or multiply objects of different types. In some cases this will evaluate to a specific type, in other cases, this will raise a `TypeError`. See the table below for details.

one operand	other operand	addition result type	multiplication result type
bool	bool	int	int
bool	str	TypeError	str
bool	int	int	int
str	str	str	TypeError
str	int	TypeError	str
int	int	int	int

For example:

```
True + False == 1
True * False == 0
True + 5 == 6
5 * False == 0
False + 'a' # TypeError
True * 'a' == 'a'
```

```
'a' + 'b' == 'ab'
'a' * 'b' # TypeError
'a' + 5 # TypeError
'a' * 5 == 'aaaaa'
1 + 2 == 3
2 * 3 == 6
```

Implement the `BinOp.check_return_type` method on the next page that checks whether an intance of `BinOp` will raise a `TypeError` when `BinOp.evaluate` is called. `BinOp.check_return_type` should not call any `Statement.evaluate` methods.

You do not need to write anything on this page. Please answer the question on the **next** page

```
def check_return_type(self) -> Optional[str]:
    """ Return a string representing the type of object that
    will be returned by calling self.evaluate.

    Return None, if self.evaluate will raise a TypeError.

    If this method returns a string rather than None, it will be one of:
    'str' or 'int'

    === Precondition ===
    self.right and self.left do not contain any Name expressions

    >>> BinOp(Bool(True), '+', Num(5)).check_return_type()
    'int'
    >>> BinOp(BinOp(Num(4), '*', Str('a')), '+', Str('b')).check_return_type()
    'str'
    >>> BinOp(BinOp(Str('a'), '*', Str('a')), '+', Num(1)).check_return_type()
    None
    """
```

8. **[11 marks] Recursive Sorting Algorithms**.

Quicksort3 is a version of quicksort that selects two pivots, partitions a list into 3 sections, and makes a recursive call on each. See a partial implementation of the non-mutating version of the `quicksort3` function below.

`_partition3` works exactly like the `_partition` helper we saw in class except it returns 3 lists: one where all values are `<` pivot1, one where all values are `>=` pivot1 and `<` pivot2, and one where all values are `>=` pivot2.

(a) **[2 marks]** Fill in the base case for the `quicksort3` function below.

```
def quicksort3(lst: List) -> List:
    if               :




    else:
        pivot1, pivot2 = lst[:2]
        small, mid, big = [quicksort3(x) for x in _partition3([2:], pivot1, pivot2)]
        return small + [pivot1] + mid + [pivot2] + big
```

(b) **[1 mark]** If we call `quicksort` on the following list:

```
[1, 4, 9, 6, 3, 7, 8, 5, 2]
```

how many times will `quicksort` be called? If we call `quicksort3` on the same list, how many times will `quicksort3` be called?
quicksort:

quicksort3:

(c) **[2 marks]** If we have the following lists:

```
lst1 = [1, 2, 3, 4, 5]
lst2 = [5, 4, 3, 2, 1]
```

Do we expect that calling `mergesort` on `lst1` should take more, less, or approximately the same amount of time as than calling `mergesort` on `lst2`.

More Less Same

Briefly explain why:

(d) **[2 marks]** Timsort has shown us that we can reduce the number of recursive calls a mergesort function has to make by finding runs that already exist in the data. If we did the same for quicksort, would finding runs also reduce the number of recursive calls?

Yes No

Briefly explain why:

(e) [**4 marks**] Here is an implementation of a merge sort algorithm that works in place (docstrings omitted) and a partial definition of the `_in_place_merge` method which merges part of a list in place:

```
def in_place_mergesort(lst: List) -> None:
    _mergesort_helper(lst, 0, len(lst))

def _in_place_merge(lst: List, small_i: int, big_i: int) -> None:
    """ Merge the section of <lst> between <small_i> and <big_i> so that
    the section of <lst> between <small_i> (inclusive) and <big_i> (exclusive)
    becomes sorted.

    === Precondition ===
    The first half of the section of <lst> between <small_i> and <big_i> is
    sorted and so is the second half of <lst> between <small_i> and <big_i>

    >>> lst = [1, 4, 6, 3, 5, -10, -5]
    >>> _inplace_merge(lst, 1, 5)
    >>> lst
    [1, 3, 4, 5, 6, -10, -5]
    """
```

Implement the `_mergesort_helper` function below which does the work of recursively sorting `lst`. This function should work exactly like the mergesort we saw in class but it should mutate the `lst` instead of returning a copy of the sorted `lst`. This function should call `_in_place_merge` as a helper. You do not need to write a docstring for this function.

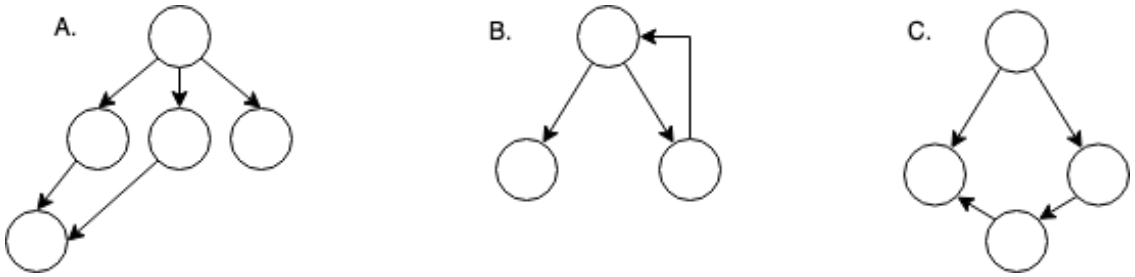
```
def _mergesort_helper(lst: List, small_i: int, big_i: int) -> None:
```

9. **[8 marks] Trees**.

(a) **[8 marks]** A `Tree` is a well formed if:

- it is a collection of nodes where each node has zero or more child nodes that it references
- every node except the root node is the child of exactly one other node
- no node is the parent of the root node

For example, the following are not well formed:



Implement the `Tree.well_formed` method that checks if a `Tree` is well formed given the definition above. The `Tree.well_formed` method must call the `Tree._well_formed_helper` method. You may add more parameters to the helper method if you wish. Make sure you add all necessary annotations to parameters and the return type. You do not need to write a docstring for either method.

```
def _well_formed_helper(self,                                    ) ->                    :
```

```
def well_formed(self) -> bool:
  """ Return True if <self> is well formed """
```

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question.*

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question.*

Basic operators

```
True and False, True or False, not True
1 + 3, 1 - 3, 1 * 3
5 / 2 == 2.5, 5 // 2 == 2, 5 % 2 == 1
'hi' + 'bye'           # 'hibye'
[1, 2, 3] + [4, 5, 6]  # [1, 2, 3, 4, 5, 6]
```

List methods

```
lst = [1, 2, 3]
len(lst)            # 3
lst[0]              # 1
lst[0:2]            # [1, 2]
lst[0] = 'howdy'    # lst == ['howdy', 2, 3]
lst.append(29)      # lst == ['howdy', 2, 3, 29]
lst.pop()           # lst == ['howdy', 2, 3], returns 29
lst.pop(1)          # lst == ['howdy', 3], returns 2
lst.insert(1, 100)  # lst == ['howdy', 100, 3]
lst.extend([4, 5])  # lst == ['howdy', 100, 3, 4, 5]
lst.remove(100)     # lst == ['howdy', 3, 4, 5]
3 in lst            # returns True
```

Dictionary methods

```
d = {'hi': 4, 'bye': 100}
list(d.keys())    # ['hi', 'bye'] # order not guaranteed
list(d.values())  # [100, 4] # order not guaranteed
d['hi']           # 4
d[100]            # raises KeyError!
'hi' in d         # True
4 in d            # False
d['howdy'] = 15   # adds new key-value pair
d['hi'] = -100    # changes a key-value pair
```

Exceptions

```
class MyCustomError(Exception):
    # Override __str__ to customize the error message.

raise MyCustomError
```

Stacks and Queues

```
s = Stack()
s.is_empty()
s.push(10)
s.pop()  # Raises an EmptyStackError if stack is empty.

q = Queue()
q.is_empty()
q.enqueue(10)
q.dequeue()  # Returns None if queue is empty.
```

A Few Builtin Functions

```
any([True, False, False])  # evaluates to True
all([True, False, False])  # evaluates to False
all([True, True, True])    # evaluates to True
sum([1, 2, 3])             # evaluates to 6
sum([1, 2, 3], 10)         # evaluates to 16
```

Classes

```
class Point:
    x: float
    y: float

    def __init__(self, x: float, y: float) -> None:
        self.x = x
        self.y = y

    def size(self) -> float:
        return (self.x ** 2 + self.y ** 2) ** 0.5

p = Point(3, 4)   # create an instace
p.x               # attribute access: evaluates to 3
p.size()          # method call: returns 5.0

class MyWeirdClass(Point):  # inheritance
    pass
```

Linked Lists

```
class _Node:
    """A node in a linked list.

    === Attributes ===
    item: The data stored in this node.
    next:
        The next node in the list, or None if there are
        no more nodes in the list.
    """
    item: Any
    next: Optional[_Node]

    def __init__(self, item: Any) -> None:
        """Initialize a new node storing <item>,
        with no 'next' node.
        """

class LinkedList:
    """A linked list implementation of the List ADT.
    """
    # === Private Attributes ===
    # _first:
    #     The first node in the linked list,
    #     or None if the list is empty.
    _first: Optional[_Node]

    def __init__(self, items: list) -> None:
        """Initialize a linked list with the given items.

        The first node in the linked list contains the
        first item in <items>.
        """
```

Comprehensions

```
cmp = [i + 2 for i in range(4)]      # cmp == [2, 3, 4, 5]
cmp = {len(x) for x in [(1,2), (4)]} # cmp == {2, 1}
```

Trees

```
class Tree:
    # === Private Attributes ===
    # _root: The item stored at this tree's root, or None if this tree is empty.
    _root: Optional[Any]
    # _subtrees: The list of all subtrees of this tree.
    _subtrees: List[Tree]

    # === Representation Invariants ===
    # - If self._root is None then self._subtrees is an empty list.
    #   This setting of attributes represents an empty tree.
    # - self._subtrees does not contain any empty trees.

    def __init__(self, root: Optional[Any], subtrees: List[Tree]) -> None:
        """Initialize a new Tree with the given root value and subtrees.

        If <root> is None, the tree is empty.
        Precondition: if <root> is None, then <subtrees> is empty.
        """

    def is_empty(self) -> bool:
        """Return whether this tree is empty."""
```

Binary Search Trees

```
class BinarySearchTree:
    # === Private Attributes ===
    # _root: The item stored at the root of the tree, or None if the tree is empty.
    _root: Optional[Any]
    # _left: The left subtree, or None if the tree is empty.
    _left: Optional[BinarySearchTree]
    # _right: The right subtree, or None if the tree is empty.
    _right: Optional[BinarySearchTree]

    # === Representation Invariants ===
    #  - If self._root is None, then so are self._left and self._right.
    #    This represents an empty BST.
    #  - If self._root is not None, then self._left and self._right are BinarySearchTrees.
    #  - (BST Property) If self is not empty, then
    #    all items in self._left are <= self._root, and
    #    all items in self._right are >= self._root.

    def __init__(self, root: Optional[Any]) -> None:
        """Initialize a new BST containing only the given root value.

        If <root> is None, initialize an empty tree.
        """
```

Abstract Syntax Trees

```
class Statement:
    """An abstract class representing
       a Python statement."""
    def evaluate(self, env: Dict[str, Any]) -> Optional[Any]:
        """Evaluate this statement
           with the given environment."""

class Expr(Statement):
    """An abstract class representing a
       Python expression."""

class BinOp(Expr):
    """An arithmetic binary operation.
    === Representation Invariants ===
    - self.op == '+' or self.op == '*'
    """
    left: Expr
    op: str
    right: Expr

    def __init__(self, left: Expr, op: str,
                 right: Expr) -> None:
        """Initialize a new binary operation expression.

        Precondition: <op> is the string '+' or '*'.
        """

    def evaluate(self, env: Dict[str, Any]) -> Any:
        """Return the *value* of this expression.

        The returned value should the result of how
        this expression would be evaluated by the Python
        interpreter.

        >>> expr = BinOp(Num(10.5), '+', Num(30))
        >>> expr.evaluate()
        40.5
        """

class Num(Expr):
    """A numeric constant literal.

    === Attributes ===
    n: the value of the constant
    """
    n: Union[int, float]

    def __init__(self, number: Union[int, float]) -> None:
        """Initialize a new numeric constant."""

    def evaluate(self, env: Dict[str, Any]) -> Any:
        """Return the *value* of this expression.

        The returned value should the result of how this
        expression would be evaluated by the Python interpreter.

        >>> expr = Num(10.5)
        >>> expr.evaluate()
        10.5
        """
```

```
class Name(Expr):
    """A variable name.

    === Attributes ===
    id: The variable name in this expression.
    """
    id: str

    def __init__(self, id_: str) -> None:
        """ Initialize a new variable """

    def evaluate(self, env: Dict[str, Any]) -> Any:
        """Return the *value* of this expression.

        The name should be looked up in the `env`
        argument to this method. Raise a NameError
        if the name is not found.

        >>> expr = Name('x')
        >>> expr.evaluate({'x': 10})
        10
        """

class Bool(Expr):
    """ Same as Num but stores a boolean literal """

class Str(Expr):
    """ Same as Num but stores a string literal """
```