CSC148 - Expression Trees: The Variable Environment

We have just introduced the concept of the **(variable) environment**, which is used to associate variable names to values in a program. To model this with our expression trees, we have to modify our abstract `Expr` class to the following:

```
class Expr:
    """An abstract class representing a Python expression."""
    def evaluate(self, env: Dict[str, Any]) -> Any:
        """Return the *value* of this expression.

        The given `env` is used to lookup variables.
        """
        raise NotImplementedError
```

The `Expr` subclasses from this week's prep don't really use this new `env` parameter, but on this worksheet we'll look at different kinds of Python expressions and statements that do!

1. Read through the following class `Name`, which represents a variable name. Then, complete its *evaluate* method, which requires looking up the variable name in the new `env` parameter.

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

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

    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
        """
```

2. A natural question to ask is "when we call `evaluate`, how do we know what `env` to pass in?" As a first step towards answering this question, we'll look at how to model *assignment statements* as a way to mutate an environment. Read through the following class, and then on the next page implement its `evaluate` method.

```
class Assign(Statement):
    """An assignment statement with a single target, like `x = 10 + 3`.

    === Attributes ===
    target: the variable name on the left-hand side of the equals sign.
    value: the expression on the right-hand side of the equals sign.
    """
    target: str
    value: Expr
```

```
    def evaluate(self, env: Dict[str, Any]) -> Optional[Any]:
        """Evaluate this statement.

        This does the following: evaluate the right-hand side expression,
        and then mutate <env> to store a binding between this statement's
        target and the corresponding value.

        >>> stmt = Assign('x', BinOp(Num(10), '+', Num(3)))
        >>> env = {}
        >>> stmt.evaluate(env)
        >>> env['x']
        13
        """
```

3. Next, we'll extend the previous class to support *parallel assignment*. Read through the following class, and implement its `evaluate` method.

```
class ParallelAssign(Statement):
    """A parallel assignment statement.

    === Attributes ===
    targets: the variable names being assigned to---the left-hand side of the =
    values: the expressions being assigned---the right-hand side of the =
    """
    targets: List[str]
    values: List[Expr]

    def evaluate(self, env: Dict[str, Any]) -> Optional[Any]:
        """Evaluate this statement.

        This does the following: evaluate each expression on the right-hand side
        and then bind each target to its corresponding value.

        Raise a ValueError if the lengths of self.targets and self.values are not equal.

        >>> stmt = ParallelAssign(['x', 'y'],
        ...                       [BinOp(Num(10), '+', Num(3)), Num(-4.5)])
        >>> env = {}
        >>> stmt.evaluate(env)
        >>> env['x']
        13
        >>> env['y']
        -4.5
        """
```