CSC111 Lecture 12: Abstract Syntax Trees, Continued

Hisbaan Noorani

February 24, 2021

Contents

Note: if you are working in PyCharm, you can use the starter file lecture12.py on Quercus for this exercise.

1 Exercise 1: If statements

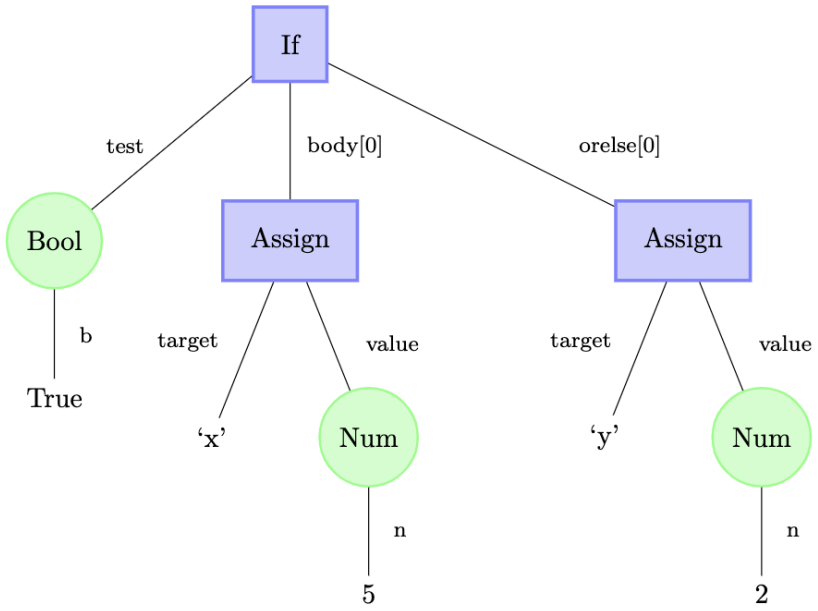
Now that you've seen the `Module` class in lecture, you are ready to implement *compound statements* whose bodies consist of multiple statements. In this exercise, we'll cover if statements.

First, review this new class to represent if statements.

```
class If(Statement):
    """An if statement.

    This is a statement of the form:

        if <test>:
            <body>
        else:
            <orelse>

    Instance Attributes:
        - test: The condition expression of this if statement.
        - body: A sequence of statements to evaluate if the condition is True.
        - orelse: A sequence of statements to evaluate if the condition is False.
    """
    test: Expr
    body: list[Statement]
    orelse: list[Statement]

    def __init__(self, test: Expr, body: list[Statement],
                 orelse: list[Statement]) -> None:
```

```
22          self.test = test
23          self.body = body
24          self.orelse = orelse
```

1. Write an AST expression using `If` (and other AST types) to represent the following diagram.




```
>>> If(Bool(b),
...    [Assign('x', Num(1))],
...    [Assign('y', Num(0))])
```

2. Write an AST expression using `If` (and other AST types) to represent the following Python code.

```
if x < 100:
    print(x)
else:
    y = x + 2
    x = 1
```

```
>>> If(Compare(Name('x'), [('<', Num(100))]),
...    [Print(Name('x'))],
...    [Assign('y', BinOp(Name('x'), '+', Num(2))), Assign('x', Num(1))]
...    )
```

3. In Python, the else branch is optional. How would we represent an if statement with no `else` branc using our `If` class?

   We can use an empty list for the `orelse` branch.

4. Implement the `If.evaluate` method. Note that you use can use if statements in your implementation (similar to how we used the `+` and `*` operators to implement `BinOp`).

```
class If:
    def evaluate(self, env: Dict[str, Any]) -> None:
        """Evaluate this statement.

        Preconditions:
            - self.test evaluates to a boolean

        >>> stmt = If(Bool(True),
        ...           [Assign('x', Num(1))],
        ...           [Assign('y', Num(0))])
        ...
        >>> env = {}
        >>> stmt.evaluate(env)
        >>> env
        {'x': 1}
        """
        test_val = self.test.evaluate(env)

        if test_val:
            for statement in self.body:
                statment.evaluate(env)
        else:
            for statement in self.orelse:
                statment.evaluate(env)
```

2 Exercise 2: For loops (over ranges)

Please take a moment to review the `ForRange` class we introduced in lecture.

```
class ForRange(Statement):
    """A for loop that loops over a range of numbers.

        for <target> in range(<start>, <stop>):
            <body>

    Instance Attributes:
        - target: The loop variable.
        - start: The start for the range (inclusive).
        - stop: The end of the range (this is *exclusive*, so <stop> is not included
                in the loop).
```

```
        - body: The statements to execute in the loop body.
    """
    target: str
    start: Expr
    stop: Expr
    body: list[Statement]

    def __init__(self, target: str, start: Expr, stop: Expr,
                 body: list[Statement]) -> None:
        """Initialize a new ForRange node."""
        self.target = target
        self.start = start
        self.stop = stop
        self.body = body
```

1. Write the Python code that corresponds to the following AST expression.

```
>>> ForRange('x',
...          Num(1),
...          BinOp(Num(2), '+', Num(3)),
...          [Print(Name('x'))])
```

```
for x in range(1, 2 + 3):
    print(x)
```

2. Write the `Module` AST expression that corresponds to the following Python code:

```
sum_so_far = 0

for n in range(1, 10):
    sum_so_far = sum_so_far + n

print(sum_so_far)
```

Hint: Your `Module` body should have three elements (an `Assign`, a `ForRange`, and a `Print`).

```
>>> Module([
...     Assign('sum_so_far', Num(0)),
...     ForRange('n', Num(1), Num(10),
...              [Assign('sum_so_far', BinOp(Name('sum_so_far'), '+', Name('n')))]),
...     Print(Name('x'))
... ])
```

3. Finally, implement the `ForRange.evaluate` method. Think carefully about how you make the loop variable accessible when you evaluate the statements in the loop body—you'll need to update the variable environment. Note that you use can use for loops in your implementation!

```python
class ForRange:
    def evaluate(self, env: Dict[str, Any]) -> None:
        """Evaluate this statement.

        Preconditions:
            - self.start and self.stop evaluate to integers

        >>> statement = ForRange('x', Num(1), BinOp(Num(2), '+', Num(3)),
        ...                      [Print(Name('x'))])
        >>> statement.evaluate({})
        1
        2
        3
        4
        """
        start_val = self.start.evaluate(env)
        stop_val = self.stop.evaluate(env)

        for i in range(start_val, stop_val):
            # Need to assign self.target to have value i
            # NOTE: only use one of the two versions

            # Version 1:
            env[self.target] = i

            # Version 2:
            Assign(self.target, Num(i)).evaluate(env)

            for statement in self.body:
                statement.evaluate(env)
```

3 Additional exercises

1. Let's add some *type-checking* to the methods we implemented in this exercise.

   (a) Modify the `If.evaluate` method to raise a `TypeError` if its condition doesn't evaluate to a boolean. (This is technically stricter than real Python.)

```python
class If:
    def evaluate(self, env: Dict[str, Any]) -> None:
        """Evaluate this statement.

        Preconditions:
            - self.test evaluates to a boolean

        >>> stmt = If(Bool(True),
```

```
        ...             [Assign('x', Num(1))],
        ...             [Assign('y', Num(0))])
        ...
        >>> env = {}
        >>> stmt.evaluate(env)
        >>> env
        {'x': 1}
        """
        test_val = self.test.evaluate(env)

        if not isinstance(test_val, bool):
            raise TypeError

        if test_val:
            for statement in self.body:
                statment.evaluate(env)
        else:
            for statement in self.orelse:
                statment.evaluate(env)
```

(b) Modify the `ForRange.evaluate` method to raise a `TypeError` if its start or stop expressions don't evaluate to an integer.

2. Modify the `If` class to support an arbitrary number of `elif` branches. You may add additional instance attributes.

3. Design and implement a new AST `Statement` subclass to represent a while loop.

4. Design and implement a new AST `Expr` subclass to represent a list comprehension over a range of numbers.