Computer Systems Modeling and Verification
(USEEN1)

Part I: Dynamic typing

Python is a dynamically typed language, which means that the python interpreter does not enforce types at compile-time, only at run-time. Moreover, variables can be reassigned to values[1] of different types during run-time, and function arguments can accept values of different types.

On the other hand, values are typed, and type checking is still performed for primitive operations at run-time: this is called *dynamic typing*.

1 Mixing integer and string operations

Try to evaluate the following expressions with Python:

```
100 + 22        100 - 22        100 * 22
"a" + "b"       "a" - "b"       "a" * "b"
100 + "b"       100 - "b"       100 * "b"
"a" + 22        "a" - 22        "a" * 22
```

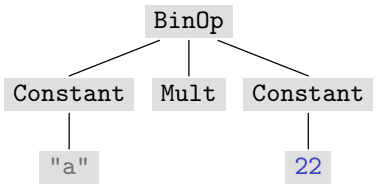
Some of the above expressions will fail to evaluate and raise a `TypeError`.

Let us now consider the fragment of Python AST for integer and string operations, with integer and string constants. Such a fragment is similar to the AST for simple arithmetic expressions, but extended with string constants such as `Constant("a")`.

For instance, you can define the AST for `"a" * 22` like this:

```
BinOp(
    Constant("a"),
    Mult(),
    Constant(22))
```

This AST could also be displayed as the following drawing:



If we want to evaluate this kind of expressions, we need to define a type for values (where values can be either integers or strings):

```
type Value = IntValue | StrValue

@dataclass
class IntValue:
    value: int

@dataclass
class StrValue:
    value: str
```

1. A value is the result of an evaluation (which cannot thus be evaluated further). A constant is a value, but a list of values is also a value, a tuple of values is also a value, and so on.

Exercises

1. Define a function `eval_expr(e: expr, m: Mapping[str, Value] = {}) -> Value` which computes an `expr` and returns its value.

2. Modify `eval(e: Expression, m: Mapping[str, Value] = {}) -> int | str` in order to return the result contained in the value.

2 Mixing integer, string and boolean operations

Try to evaluate the following expressions with Python:

```
True + False        True - False        True * False
True + "b"          True - "b"          True * "b"
True + 22           True - 22           True * 22
 "a" + False         "a" - False         "a" * False
 100 + False         100 - False         100 * False
```

Some of the above expressions will fail to evaluate and raise a `TypeError`.

Let us now consider the fragment of Python AST for integer, string and boolean operations, with integer, string and boolean constants. Such a fragment is similar to the previous AST, but extended with boolean constants such as `Constant(True)`.

If we want to evaluate this kind of expressions, we need to extends the type for values (where values can be either integers or strings or booleans):

```
type Value = IntValue | StrValue | BoolValue

@dataclass
class BoolValue:
    value: bool

# ...
```

Exercises

1. Modify `eval_expr(e: expr, m: Mapping[str, Value] = {}) -> Value` in order to deal with boolean values.

2. Modify `eval(e: Expression, m: Mapping[str, Value] = {}) -> int | str | bool` in order to deal with boolean values.

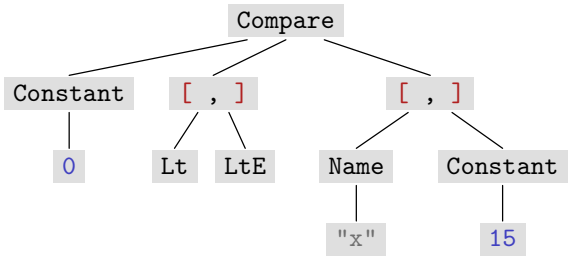
2.1 Adding comparison operators

Python supports chaining of comparison operators. For instance, you can write `0 < x <= 15`, which corresponds to the expression `0 < x and x <= 15`.

In the AST of a comparison, you shall thus find the first expression, a list of comparison operators and a list of comparators. For instance, you can define the AST for `0 < x <= 15` like this:

```
Compare(
    Constant(0),
    [Lt(), LtE()],
    [Name("x"), Constant(15)])
```

This AST could also be displayed as the following drawing:

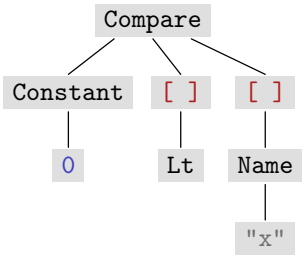


In the following, we will consider only usual binary comparisons. Of course, in the AST you will still find a first expression, then a list of comparison operators and finally a list of comparators. However, we will asume that those two lists always contain only one element.

For instance, you can define the AST for `0 < x` like this:

```
Compare(
    Constant(0),
    [Lt()],
    [Name("x")])
```

This AST could also be displayed as the following drawing:



We will consider only the following (binary) comparisons operators:

`Eq` (`==`), `NotEq` (`!=`), `Gt` (`>`), `Lt` (`<`), `GtE` (`>=`), `LtE` (`<=`)

Exercises

1. Modify `eval_expr(e: expr, m: Mapping[str, Value] = {}) -> Value` in order to deal with these comparisons operators.

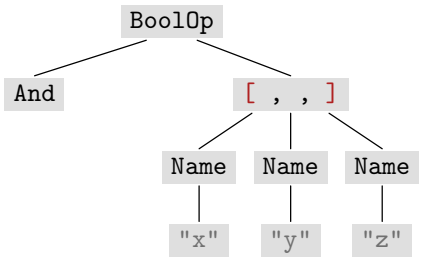
2.2 Adding logical operators

Python also supports chaining of logical operators: consecutive operations with the same operator, such as `x and y and z`, are collapsed into one `And` with several values.

For instance, you can define the AST for `x and y and z` like this:

```
BoolOp(
    And(),
    [Name("x"), Name("y"), Name("z")])
```

This AST could also be displayed as the following drawing:

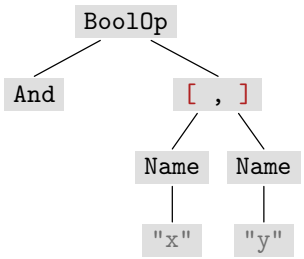


In the following, we will consider only the usual binary logical operators (in other words, we assume that boolean expressions are always parenthesized). Of course, in the AST you will still find first the operator and then a list of expressions. However, we will assume that this list always contain only two element.

For instance, you can define the AST for `x and y` like this:

```
BoolOp(
    And(),
    [Name("x"), Name("y")])
```

This AST could also be displayed as the following drawing:



We will consider only the following (binary) logical operators:

`And` (`and`), `Or` (`or`)

Exercises

1. Modify `eval_expr(e: expr, m: Mapping[str, Value] = {}) -> Value` in order to deal with these boolean operators.

3 Adding user-defined functions

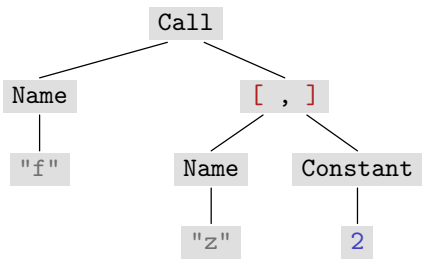
Let us consider simple user-defined functions such as the following one:

```
def f(x, y):
    return x + y
```

Such a function can then be called inside an expression. For instance, you can define the AST for `f(z, 2)` like this:

```
Call(
    Name("f"),
    [Name("z"), Constant(2)])
```

This AST could also be displayed as the following drawing:



The function itself is considered a value (since it cannot be evaluated further without actual arguments). We thus need to extend the type of values:

```
type Value = IntValue | StrValue | BoolValue | FunValue

@dataclass
class FunValue:
    args: list[str]
    body: expr

# ...
```

You can then create a value for function `f` as follows:

```
f = ast.parse("x␣+␣y", mode='eval')
v = FunValue(["x", "y"], f.body)
```

Note. The types of the formal arguments are not available in `FunValue`, and the types of the actual arguments cannot thus be (dynamically) checked in a function call. In the Python interpreter, these types are not checked either (even if they are explicit in the source code). However, the interpreter does check that the number of actual arguments is correct.

Exercises

1. Modify `eval_expr(e: expr, m: Mapping[str, Value] = {}) -> Value` in order to deal with function calls.

   You can then try the evaluation function as follows:

   ```
   src = "f(z,␣2)␣+␣1"
   e = ast.parse(src, mode='eval')
   print(eval(e, {"z": IntValue(7), "f": FunValue(["x", "y"], f.body)}))
   ```