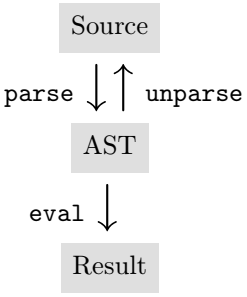
Computer Systems Modeling and Verification
(USEEN1)

Part I: Abstract Syntax Trees

Abstract Syntax Trees (AST) are tree representations of source codes. They are used in every interpreter, as summarized in the following picture:



The *parse* function converts the source code into a tree structure. It is usually also be possible to *unparse* an AST into source code form.

Most tools that perform some kind of static analysis of the code (such as linters or type checkers) actually work on the AST.

1 Arithmetic expressions without variables

The definition of Python AST for simple arithmetic expressions (without variables) is similar to what follows (the actual definition is slightly different in order to be compatible with older Python versions).

First, there is an enumerated type for binary operators similar to the following one (with the corresponding empty classes):

```
type operator = Add | Sub | Mult # ...
```

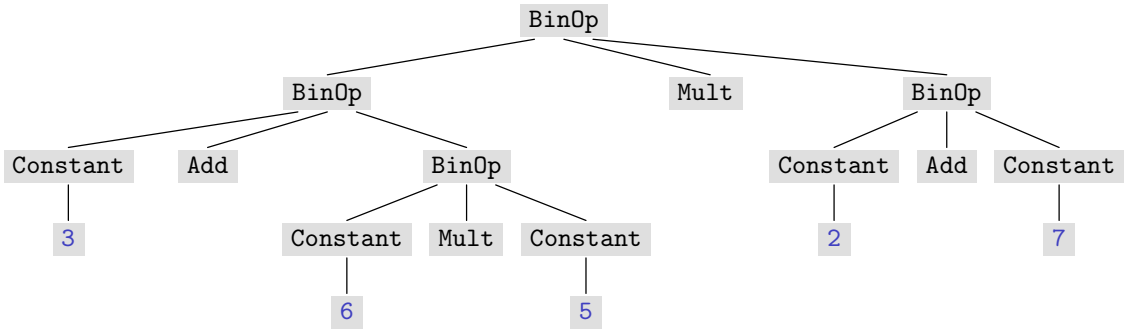
The type for arithmetic expressions is then similar to the following one:

```
type expr = Constant | BinOp # ...

@dataclass
class Constant:
    value: int # ...

@dataclass
class BinOp:
    left: expr
    op: operator
    right: expr
```

For instance, you can define the AST for `(3 + (6 * 5)) * (2 + 7)` like this:

```
BinOp(
    BinOp(
        Constant(3),
        Add(),
        BinOp(
            Constant(6),
            Mult(),
            Constant(5))),
    Mult(),
    BinOp(
        Constant(2),
        Add(),
        Constant(7)))
```

This AST could also be displayed as the following drawing:



Exercises

1. Define a function `display_inline(e: expr) -> str` which displays an expression as a string on a single line (similarly to what the `print` function does).
2. Define a function `display(e: expr, indent: int, depth: int = 0) -> str` which displays an expression as a string as in the example above (using the `depth` parameter as indentation level).
3. Define a function `dump(e: expr, indent: int | None = None) -> str` which displays an expression as a string using `display_inline` if indent is `None` and `display` otherwise.
4. Define a function `unparse(e: expr) -> str` which displays an `expr` as a string (using the usual syntax with infix operators).
5. Define a function `eval_expr(e: expr) -> int` which computes an `expr` and returns its integer value.

2 Arithmetic expressions with variables

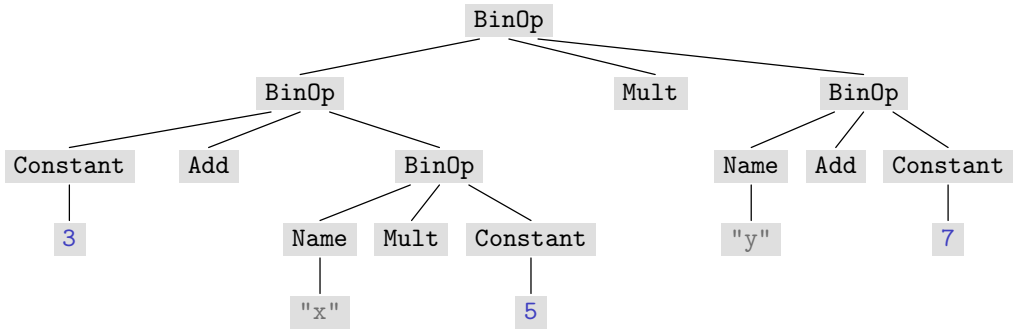
Let us add variables to type `expr` in order to account for variables in expressions:

```
type expr = Name | Constant | BinOp # ...

@dataclass
class Name:
    id: str

# ...
```

For instance, you can define the AST for `(3 + (x * 5)) * (y + 7)` like this:

```
BinOp(
    BinOp(
        Constant(3),
        Add(),
        BinOp(
            Name("x"),
            Mult(),
            Constant(5))),
    Mult(),
    BinOp(
        Name("y"),
        Add(),
        Constant(7)))
```

This AST could also be displayed as the following drawing:



Exercises

1. Define and check that the given example is well typed. Test your code by printing this expression (using the standard `print` function).

2. Modify function `display_inline(e: expr) -> str` in order to deal with variables.

3. Modify function `display(e: expr, indent: int, depth: int = 0) -> str` in order to deal with variables.

4. Check that function `dump(e: expr, indent: int | None = None) -> str` still works as expected.

5. Modify function `unparse(e: expr) -> str` in order to deal with variables.

6. Modify function `eval_expr(e: expr) -> int` in order to deal with variables. You need add a new parameter to provide the value of variables. A mapping is well suited for this purpose, and the new prototype is thus:

   ```
   eval_expr(e: expr, m: Mapping[str, int]) -> int
   ```