```
import ast
import operator
from typing import List, Dict

from base_node_class import BaseOperationNodeClass
from calculator import Calculator


class Name(BaseOperationNodeClass):
    """
        The Name class is a subclass of the BaseOperationNodeClass and represents the variable name node in the computation
        tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Name node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Variable name implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.id


class Num(BaseOperationNodeClass):
    """
        The Name class is a subclass of the BaseOperationNodeClass and represents the numeral node in the computation tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Num node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
```

```
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Numeral implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.n


class Binop(BaseOperationNodeClass):
    """
        The Name class is a subclass of the BaseOperationNodeClass and represents the binary operation node in the
        computation tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Binop node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    """
        Mapping between ast mathematical operations and the string name of the operation.
    """
    _OP_NAME_MAP: Dict[type(ast), str] = {
        ast.Add: "add",
        ast.Sub: "sub",
        ast.Mult: "mult",
        ast.Div: "div",
        ast.Invert: "neg"
    }

    """
        Mapping between the string name of the operation and its symbol.
    """
    _OP_SYM_MAP: Dict[str, str] = {
        "add": "+",
        "sub": "-",
        "mult": "*",
        "div": "/",
        "neg": "-"
```

```
    }

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Binary operation implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return self._OP_NAME_MAP[type(ast_node.op)]

    def generate_op_sym(self) -> str:
        """
        Generates the symbol of the mathematical operation out of the operation string (e.g. add, sub, ..).
        :returns generated symbol
        """
        return self._OP_SYM_MAP[self.name]


class Call(BaseOperationNodeClass):
    """
        The Call class is a subclass of the BaseOperationNodeClass and represents the function calls (e.g. sin/cos,..) node
        in the computation tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Function (call) node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Function call implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.func.id


class Output(BaseOperationNodeClass):
    """
        The Output class is a subclass of the BaseOperationNodeClass and represents the output node in the computation tree.
```

```
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Output node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Output implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.targets[0].id


class Subscript(BaseOperationNodeClass):
    """
        The Subscript class is a subclass of the BaseOperationNodeClass and represents the array field access node in the
        computation tree.
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Subscript node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)
        # initialize local fields
        self.index: List[int] = list()
        self.create_index(ast_node)

    """
        Mapping between the index of the operation and its position (actually always 0).
    """
    _VAR_MAP: Dict[str, int] = {
        "i": 0,
        "j": 0,
```

```
        "k": 0
    }

    """
        Mapping between the operation and its symbol.
    """
    _OP_SYM_MAP: Dict[type(ast), str] = {
        ast.Add: "+",
        ast.Sub: "-"
    }

    def create_index(self,
                     ast_node: ast) -> None:
        """
        Create the numerical index of the array field access e.g. convert [i+2, j-3, k] to [2,-3,0]
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        # create index
        self.index = list()
        for slice in ast_node.slice.value.elts:
            if isinstance(slice, ast.Name):
                self.index.append(self._VAR_MAP[slice.id])
            elif isinstance(slice, ast.BinOp):
                # note: only support for index variations [i, j+3,..]
                # read index expression
                expression = str(slice.left.id) + self._OP_SYM_MAP[type(slice.op)] + str(slice.right.n)
                # convert [i+1,j, k-1] to [1, 0, -1]
                calculator = Calculator()
                self.index.append(calculator.eval_expr(self._VAR_MAP, expression))

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Subscript (array field access) implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return ast_node.value.id

    def generate_label(self) -> str:
        """
        Subscript (array field access) implementation of generate_label.
        :returns generated label
        """
        return str(self.name) + str(self.index)


class Ternary(BaseOperationNodeClass):
```

```
    """
        The Ternary operator class is a subclass of the BaseOperationNodeClass and represents ternary operation of the
        form: expression_true if comparison_expression else expression_false
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Ternary node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # initialize superclass
        super().__init__(ast_node, number)

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Ternary operator implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return "?"


class Compare(BaseOperationNodeClass):
    """
        The Comparison operator class is a subclass of the BaseOperationNodeClass and represents the comparison of two
    """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new Compare node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # set comparison operator field
        self.op: operator = self._COMP_MAP[type(ast_node.ops[0])]
        # initialize superclass
        super().__init__(ast_node, number)

    """
        Mapping between the abstract syntax tree (python) comparison operator and the operator comparison operator.
    """
    _COMP_MAP: Dict[type(ast), type(operator)] = {
        ast.Lt: operator.lt,
```

```
        ast.LtE: operator.le,
        ast.Gt: operator.gt,
        ast.GtE: operator.ge,
        ast.Eq: operator.eq
    }

    """
        Mapping between the operator comparison operator and its mathematical string symbol.
    """
    _COMP_SYM: Dict[type(operator), str] = {
        operator.lt: "<",
        operator.le: "<=",
        operator.gt: ">",
        operator.ge: ">=",
        operator.eq: "=="
    }

    def generate_name(self,
                      ast_node: ast) -> str:
        """
        Comparison operator implementation of generate_name.
        :param ast_node: abstract syntax tree node of the computation
        :returns generated name
        """
        return self._COMP_SYM[self.op]


class UnaryOp(BaseOperationNodeClass):
    """
         The UnaryOp operator class is a subclass of the BaseOperationNodeClass and represents unary operations. In our
         case we only support negation (mathematical - sign) as unary operation.
     """

    def __init__(self,
                 ast_node: ast,
                 number: int) -> None:
        """
        Create new unary operation node with given initialization parameters.
        :param ast_node: abstract syntax tree node of the computation
        :param number: tree walk numbering
        """
        # set unary operator field
        self.op: operator = self._UNARYOP_MAP[type(ast_node.op)]
        # initialize superclass
        super().__init__(ast_node, number)

    """
        Mapping between the ast unary operation and the operator operation.
    """
```

```
_UNARYOP_MAP: Dict[type(ast), type(operator)] = {
    ast.USub: operator.sub
}

"""
    Mapping between the operator unary operator and its mathematical string.
"""
_UNARYOP_SYM: Dict[type(operator), str] = {
    operator.sub: "neg"
}

"""
     Mapping between the mathematical string and its symbol.
 """
_UNARYOP_SYM_NAME = {
    "neg": "-"
}

def generate_name(self,
                  ast_node: ast) -> str:
    """
    Unary operator implementation of generate_name.
    :param ast_node: abstract syntax tree node of the computation
    :returns generated name
    """
    return self._UNARYOP_SYM[self.op]

def generate_op_sym(self) -> str:
    """
    Generates the symbol of the mathematical operation out of the operation string.
    :returns generated symbol
    """
    return self._UNARYOP_SYM_NAME[self.name]
```