Third Project: Semantic Analysis

Once syntax trees are built, additional analysis can be done by evaluating attributes on tree nodes to gather necessary semantic information from the source code not easily detected during parsing. It usually includes type checking, symbol table construction to makes sure a variable is declared before use, and decorating the AST to prepare it for the next compilation phase.

Type System

First, you will need to define objects that represent the different builtin data types and record information about their capabilities.

Let's define classes that represent types. There is a general class used to represent all types. Each basic type is then a singleton instance of the type class.

```
class MJType:
      pass

int_type = MJType("int", ...)
char_type = MJType("char", ...)
```

The contents of the type class is entirely up to you. However, you will minimally need to encode some information about what operators are supported (+, -, *, etc.), and default values.

Once you have defined the built-in types, you will need to make sure they get registered with any symbol tables or code that checks for type names.

```
class MJType:
    """
    Class that represents a type in the MiniJava language.  Basic
    Types are declared as singleton instances of this type.
    """

    def __init__(
        self, name, binary_ops=set(), unary_ops=set(),
        rel_ops=set(), assign_ops=set()
    ):
        """
        You must implement yourself and figure out what to store.
        """
        self.typename = name
        self.unary_ops = unary_ops
        self.binary_ops = binary_ops
        self.rel_ops = rel_ops
        self.assign_ops = assign_ops

    def __str__(self):
        return f"type({self.typename})"

# Create specific instances of basic types. You will need to add
# appropriate arguments depending on your definition of MJType
IntType = MJType(
    name="int",
    binary_ops={"+", "-", "*", "/", "%"},
    unary_ops={"-", "+"},
    rel_ops={"==", "!=", "<", ">", "<=", ">="},
    assign_ops={"="},
)

BooleanType = MJType(
    name="boolean",
    # TODO: Complete
)

CharType = MJType(
    name="char",
    # TODO: Complete
)

# TODO: Complete the basic types in your repository

# Array & Object types need to be instantiated for each declaration
class ArrayType(MJType):
    def __init__(self, array_type: str, element_type: str, size=None):
        """
        :param array_type: Type of the array (int[] or char[])
        :param element_type: Type of the array elements (int or char)
        :param size: Integer with the length of the array.
        """
        self.size = size
        self.element_type = element_type
        super().__init__(
            name=array_type, rel_ops={"==", "!="}, assign_ops={"="}
        )

# TODO: Complete in your repository
```

In your type checking code, you will need to reference the above type objects. Think of how you will want to access them.

Symbol table

You will need to define a symbol table that keeps track of previously declared identifiers. The symbol table will be consulted whenever the compiler needs to lookup information about variable and constant declarations.

```
class SymbolTable:
    """Class representing a symbol table.

    `add` and `lookup` methods are given, however you still need to find a way to
    deal with scopes.

    ## Attributes
    :data: the content of the SymbolTable
    """

    def __init__(self) -> None:
        """Initializes the SymbolTable."""
        self.__data = dict()

    @property
    def data(self) -> Dict[str, Any]:
        """Returns a copy of the SymbolTable."""
        return deepcopy(self.__data)

    def add(self, name: str, value: Any) -> None:
        """Adds to the SymbolTable.

        :param name: the identifier on the SymbolTable
        :param value: the value to assign to the given `name`
        """
        self.__data[name] = value

    def lookup(self, name: str) -> Union[Any, None]:
        """Searches `name` on the SymbolTable and returns the value
        assigned to it.

        :param name: the identifier that will be searched on the SymbolTable
        :return: the value assigned to `name` on the SymbolTable. If `name` is not found, `None` is returned.
        """
        return self.__data.get(name)
```

❗Notice that the semantic analysis will have to manage the symbol table in order to handle the multiple scopes of the program. To do so, you must increment the existing implementation given to you.

Try to think an answer to the following questions to implement multiple scopes:

- Which data structure can represent multiple scopes?
- When should a new scope be created?
- When should an existing scope be deleted?
- How should a variable lookup work with multiple scopes?

Visiting the AST

The visitor pattern is often used in compiler to traverse data structures that represent the programs, either syntax tree or any other intermediate representations. For this purpose, we provide the following `NodeVisitor` class to allow you to visit the AST. This class was modeled after Python's own AST visiting facilities (the AST module of Python 3).

```
class NodeVisitor:
    """ A base NodeVisitor class for visiting uc_ast nodes.
        Subclass it and define your own visit_XXX methods, where
        XXX is the class name you want to visit with these
        methods.
    """

    _method_cache = None

    def visit(self, node):
        """ Visit a node.
        """

        if self._method_cache is None:
            self._method_cache = {}

        visitor = self._method_cache.get(node.__class__.__name__, None)
        if visitor is None:
            method = 'visit_' + node.__class__.__name__
            visitor = getattr(self, method, self.generic_visit)
            self._method_cache[node.__class__.__name__] = visitor

        return visitor(node)

    def generic_visit(self, node):
        """ Called if no explicit visitor function exists for a
            node. Implements preorder visiting of the node.
        """
        for _, child in node.children():
            self.visit(child)
```

For example, a small visitor for constant can be implemented this way:

```
class ConstantVisitor(NodeVisitor):
    def __init__(self):
        self.values = []

    def visit_Constant(self, node: Constant):
        self.values.append(node.value)
```

This visitor would create a list of values of all the constant nodes encountered below the given node. To use it, simply instantiate the visitor and call its visit method on the node of your choice:

```
cv = ConstantVisitor()
cv.visit(node)
```

Note that:

- The `generic_visit()` method will be called for AST nodes for which no `visit_XXX` method was defined.
- The children of nodes for which a `visit_XXX` method was defined will not be visited - if you need this, call `generic_visit()` on the node.
- The `generic_visit()` method could be implemented more efficiently by defining AST nodes as iterable object using the `__iter__()` method of Python. Feel free to optimize

Standardized Errors

To check for semantic errors and to print their description, the following infrastructure should be used:

```
class SemanticError:
    def __init__(self, message, coord):
        self.message = message
        self.coord = coord

    def __str__(self):
        return f"SemanticError: {self.message} {self.coord}"

# Factory for creating semantic errors
class SemanticErrorFactory:
    @staticmethod
    def create(
        error_type: SE, name: str = "", ltype: str = "", rtype: str = ""
    ) -> SemanticError:
        error_msgs = {
            SE.UNDECLARED_NAME: f"{name} is not defined",
            SE.UNDECLARED_CLASS: f"{name} is not a class type",
            SE.UNDECLARED_METHOD: f"{name} is not a class method",
            SE.UNDECLARED_FIELD: f"{name} is not defined in the class",
            SE.ALREADY_DECLARED_NAME: f"Name {name} is already defined in this scope",
            SE.ALREADY_DECLARED_CLASS: f"{name} class has already been defined",
            SE.ALREADY_DECLARED_METHOD: f"{name} method has already been defined",
            SE.ALREADY_DECLARED_FIELD: f"Field {name} has already been defined",
            SE.ASSERT_EXPRESSION_TYPE_MISMATCH: "Assert expression must be of type(bool)",
            SE.PRINT_EXPRESSION_TYPE_MISMATCH: "Print expression must be of type(char), type(int) or type(String)",
            SE.ASSIGN_TYPE_MISMATCH: f"Cannot assign {rtype} to {ltype}",
            SE.ARRAY_REF_TYPE_MISMATCH: f"Cannot do an ArrayRef of {ltype}",
            SE.BINARY_EXPRESSION_TYPE_MISMATCH: f"Binary operator {name} does not have matching LHS/RHS types",
            SE.UNSUPPORTED_BINARY_OPERATION: f"Binary operator {name} is not supported by {ltype}",
            SE.WRONG_BREAK_STATEMENT: "Break statement must be inside a loop",
            SE.ARRAY_DIMENTION_MISMATCH: f"Array dimension must be of type(int), not {ltype}",
            SE.ARGUMENT_COUNT_MISMATCH: f"Number of arguments to call {name} method mismatch",
            SE.PARAMETER_TYPE_MISMATCH: f"Type mismatch with parameter {name}",
            SE.PARAMETER_ALREADY_DECLARED: f"Parameter {name} has already been defined",
            SE.CONDITIONAL_EXPRESSION_TYPE_MISMATCH: f"conditional expression is {ltype}, not type(bool)",
            SE.RETURN_TYPE_MISMATCH: f"Return of {ltype} is incompatible with {rtype} method definition",
            SE.UNSUPPORTED_UNARY_OPERATION: f"Unary operator {name} is not supported",
            SE.OBJECT_TYPE_MUST_BE_A_CLASS: f"The type of {name} must be a class",
            SE.INVALID_LENGTH_TARGET: "The target of the length operation must be of type array or String",
            SE.NOT_A_CONSTANT: "Expression must be a constant",
            SE.ARRAY_ELEMENT_TYPE_MISMATCH: f"{name} of {rtype} is incompatible with {ltype} of array",
        }

        return SemanticError(error_msgs[error_type], "")

# Semantic assert function that uses the factory
# to create and throw the exception
def assert_semantic(
    condition: bool,
    error_type: SemanticErrorType,
    coord: str,
    name: str = "",
    ltype: str = "",
    rtype: str = "",
):
    if not condition:
        semantic_error = SemanticErrorFactory.create(
            error_type, name, ltype, rtype
        )
        semantic_error.coord = coord
        print(semantic_error, file=sys.stdout)
        sys.exit(1)
```

As shown in the function code, each message is associated to a specific semantic error (SE). The error message is printed according to the given coordinate and facultative arguments. The `assert_semantic` function is used to standardize the output of the semantic analysis and must be used to pass the automatic tests.

The function checks the provided condition. If the condition is `True`, no action is taken and the semantic analysis continues. If the condition is `False`, the function uses the `SemanticErrorFactory` (defined alongside the enum) to create a semantic error object:

- The factory maps the provided `error_type` to a specific error message.
- It uses additional parameters (`name`, `ltype`, `rtype`) to format a detailed error message.

Some examples showing how to use the method are provided in the next section.

Errors Samples

Below there are examples of code that should trigger some of the errors above.

1. `x` is not defined

   ```
   int a = x;
   ```

2. Subscript must be of type(int), not `type(char)`

   ```
   int[] arr = new array[3]; int b = arr['p'];
   ```

3. Expression must be of type(bool)

```
public void f() { assert 2; };
```

4. Cannot assign `type(char)` to `type(int)`

   ```
   public void f() { int a; a = 'c'; }
   ```

5. Binary operator `>` does not have matching LHS/RHS types

   ```
   public void f (int a, char b) { if (a > b) {} }
   ```

6. Binary operator `+` is not supported by `type(char)`

   ```
   public char c = 'a' + 'b';
   ```

7. Break statement must be inside a loop

   ```
   public void f () { break; }
   ```

8. `var` initialization type mismatch

   ```
   int var = 'a';
   ```

9. var initialization must be a single element

   ```
   int var = {1, 2};
   ```

10. conditional expression is type(int), not type(bool)

    ```
    public void f () { if (2){} };
    ```

11. `i` is not a function

    ```
    public void f(int i) { i(3); }
    ```

12. No. arguments to call {name} function mismatch

    ```
    public int sum (int a, int b) { return a + b; }
    public void f() { sum(1, 2, 3); }
    ```

13. Type mismatch with parameter `b`

    ```
    public int sum (int a, int b) { return a + b; }
    public void f() { sum(1, '1'); }
    ```

14. The condition expression must be of type(bool)

    ```
    public void f() { int i = 0; if (i) {} }
    ```

15. Expression must be a constant

    ```
    public void f() { int i = 1; int y[1] = {i}; };
    ```

16. Return of `type(char)` is incompatible with `type(int)` function definition

    ```
    public int f() { return '1'; }
    ```

17. Name `i` is already defined in this scope

    ```
    public void f(int i) { int i; }
    ```

18. Unary operator `!` is not supported

```
    char c = !'a';
```

Implementing the analysis

Finally, you'll need to write code that walks the AST, decorates it with additional information and enforces a set of semantic rules as explained by the guidelines below. For walking the AST, use the NodeVisitor class.

An initial implementation of the semantic analysis is provided in the code below.

In the case of MiniJava, the language permits the use of symbols—such as classes, methods, and fields—before their actual declaration in the source code. To handle this characteristic correctly, the semantic analysis phase must be divided into two distinct stages: global symbol collection and semantic validation.

First Stage: Symbol Table Construction

In this initial stage, the compiler performs a top-level traversal of the program's abstract syntax tree (AST) to construct the global symbol table. This includes collecting and registering all class declarations, their respective fields (attributes), and method signatures (names, return types, and parameter types), without yet analyzing the internal bodies of methods. The goal is to gather enough information so that all identifiers can be resolved during the next phase, even if their declarations appear later in the code. In the implementation, this stage is handled by the `SymbolTableBuilder` class, which walks through the AST and populates a hierarchical symbol table structure representing class scopes and their members.

Second Stage: Full Semantic Analysis

With the global symbol table built, the compiler proceeds to the second stage: a complete semantic analysis of the program. In this phase, the AST is traversed again, this time to validate the correctness of type usage, ensure that variables and methods are properly declared before use, check method bodies for type consistency, and enforce language rules such as valid assignments, and control structure conditions. The `SemanticAnalyzer` class is responsible for this stage. It relies on the symbol information collected earlier to resolve identifiers and ensure that the program adheres to the MiniJava language's semantic constraints.

```
class SymbolTableBuilder(NodeVisitor):
    """Symbol Table Builder class.
    This class build the Symbol table of the program
    by visiting all the AST nodes
    using the visitor pattern.
    """

    def __init__(self):
        self.global_symtab = SymbolTable()
        self.current_class: ClassMetaData | None = None
        self.typemap = {
            "boolean": BooleanType,
            "char": CharType,
            "int": IntType,
            "String": StringType,
            "void": VoidType,
            "int[]": IntArrayType,
            "char[]": CharArrayType,
            "method": MethodType,
            "object": ObjectType,
        }

    def visit_Program(self, node: Program):
        """Visit the program node to fill in the global symbol table"""
        # First, register all classes in the program.
        # Populating the global symbol table with these classes
        for class_decl in node.class_decls:
            # Check if the class has already been declared
            # raise an ALREADY_DECLARED_CLASS semantic error if it was
            class_name = class_decl.name.name
            assert_semantic(
                error_type=SE.ALREADY_DECLARED_CLASS,
                condition=(self.global_symtab.lookup(class_name) is None),
                coord=class_decl.coord,
                name=class_name,
            )

            # Class meta data
            class_meta = ClassMetaData(
                name=class_decl.name.name,
            )
            # Add the class declaration in the global symtab
            self.global_symtab.add(class_name, class_meta)

        # Now, process each class to fill in fields and methods
```

```python
        for class_decl in node.class_decls:
            self.visit(class_decl)

        # Finally, return the global symtab to use in the next steps
        return self.global_symtab

    def visit_ClassDecl(self, node: ClassDecl):
        # Set the current class to ensure the context for internal visits
        self.current_class = self.global_symtab.lookup(node.name.name)

        # First, if the class extends another, check that the parent exists.
        # If the parent exists, then add its name in the current class metadata

        # Then, visit all fields (var_decls) of the class
        for field in node.var_decls:
            self.visit(field)

        # Finally, visit all class methods (method_decls)
        for method in node.method_decls:
            self.visit(method)

        # Unset the current class context
        self.current_class = None

    def visit_VarDecl(self, node: VarDecl):
        # First, check if the field has already been declared
        # If it was, raise an ALREADY_DECLARED_FIELD semantic error
        field_name = node.name.name
        assert_semantic(
            error_type=SE.ALREADY_DECLARED_FIELD,
            condition=(field_name not in self.current_class.fields),
            coord=node.coord,
            name=field_name,
        )

        # Now, record the field and its type
        # TODO: Use the MJTypes here?
        self.current_class.fields[field_name] = node.type.name

    def visit_MethodDecl(self, node: MethodDecl):
        # First, check if the method has already been declared
        method_name = node.name.name
        assert_semantic(
            error_type=SE.ALREADY_DECLARED_METHOD,
            condition=(method_name not in self.current_class.methods),
            coord=node.coord,
            name=method_name,
        )

        # Gather parameter types
        # Now, record the method and its signature
        # TODO: Complete

    def visit_MainMethodDecl(self, node: MainMethodDecl):
        # The main method must have the name "main"
        main_method_name = "main"
        # First, check if the main method has already been declared
        assert_semantic(
            error_type=SE.ALREADY_DECLARED_METHOD,
            condition=(main_method_name not in self.current_class.methods),
            coord=node.coord,
            name=main_method_name,
        )
        # Now, record the main method and its signature
        # TODO: Complete

class SemanticAnalyzer(NodeVisitor):
    """Semantic Analyzer class.
    This class performs semantic analysis on the AST of a MiniJava program.
    You need to define methods of the form visit_NodeName()
    for each kind of AST node that you want to process.
    """

    def __init__(self, global_symtab: SymbolTable):
        """
        :param global_symtab: Global symbol table
         with all class declaration metadata.
        """
        self.global_symtab = global_symtab
        self.typemap = {
            "boolean": BooleanType,
            "char": CharType,
            "int": IntType,
            "String": StringType,
            "void": VoidType,
            "int[]": IntArrayType,
            "char[]": CharArrayType,
            "method": MethodType,
            "object": ObjectType,
        }
        # TODO: Add symbol table and scope management

    def visit_Program(self, node: Program):
        # Visit all class declarations in the program
        for cls in node.class_decls:
            self.visit(cls)

    def visit_ClassDecl(self, node: ClassDecl):
        # Visit the fields of the class (var_decls)
        for field in node.var_decls:
            self.visit(field)

        # Then, visit the methods of the class (method_decls)
        for method in node.method_decls:
            self.visit(method)

    def visit_BinaryOp(self, node: BinaryOp):
        # Visit the left expression
        self.visit(node.lvalue)
        # Visit the right expression
        self.visit(node.rvalue)
        # Check if left and right operands have the same type
        ltype = node.lvalue.mj_type
        rtype = node.rvalue.mj_type
        assert_semantic(
            condition=(ltype == rtype),
            error_type=SE.BINARY_EXPRESSION_TYPE_MISMATCH,
            coord=node.coord,
            name=node.op,
            ltype=ltype,
            rtype=rtype,
        )
        # Check if the operation is supported by the type
        # Assign the type of the result of the binary expression
        # TODO: Complete
```

IMPORTANT: The AST you built previously only contains information (like types) at specific nodes. Beside finding the possible remaining errors of the program, the semantic analysis should be used to figure out additional information (such as typing all expressions) which will be useful for code generation, the next compilation phase. This process is usally called "decorating the AST".

Guidelines

Additionnally, we provide a set of guidelines that can be used to implement each function of the semantic analysis (type checking, definition checking, etc). Please read those carefully.

`visit_Program`

- **Action:**
  Traverse all class declarations.
- **Checks:**
  - Build the global symbol table using `SymbolTableBuilder`.
  - **Semantic Error:** If a class is declared more than once, raise `ALREADY_DECLARED_CLASS`.
- **Tip:** This is the entry point for semantic analysis; ensure that the global symbol table is correctly populated.

`visit_ClassDecl`

- **Action:**
  Process a class declaration.
- **Checks:**
  - Set the current class context.
  - If the class extends another, verify that the parent class exists.

        - **Semantic Error:** If the parent is not found, raise `UNDECLARED_CLASS`.
    - Visit each field (`VarDecl`) and method (`MethodDecl`) declaration.
- **Tip:** After processing the class, clear the current class context.

`visit_VarDecl`

- **Action:**
  Process a variable declaration.
- **Checks:**
    - Validate that the variable's type is defined.
        - **Semantic Error:** If the type is not built-in and not found in the global symbol table, raise `UNDECLARED_CLASS`.
    - Ensure that the variable is not already declared in the current scope.
        - **Semantic Error:** If already declared, raise `ALREADY_DECLARED_NAME`.
    - If an initializer is provided:
        - Visit the initializer node.
        - Check that the initializer's type is compatible with the variable's type.
            - **Semantic Error:** If types mismatch, raise `ASSIGN_TYPE_MISMATCH`.
        - For initializer lists (array initializations), verify that every element is a constant and of the correct type.
            - **Semantic Error:** If an element is not a constant or has a mismatched type, raise `ARRAY_ELEMENT_TYPE_MISMATCH`.
- **Tip:** Finally, register the variable in the current scope.

`visit_MethodDecl`

- **Action:**
  Process a method declaration.
- **Checks:**
    - Push a new scope for parameters and local variables.
    - Set the method's return type using the type map.
    - Check for duplicate method declarations.
        - **Semantic Error:** If the method is already declared, raise `ALREADY_DECLARED_METHOD`.
    - Visit the parameter list and method body.
    - Pop the scope after finishing the method body.
- **Tip:** Maintain a reference to the current method to use in return type checks later.

`visit_MainMethodDecl`

- **Action:**
  Process the main method declaration.
- **Checks:**
    - Ensure the method is named `main` and is declared only once.
        - **Semantic Error:** If the main method is already declared, raise `ALREADY_DECLARED_METHOD`.
    - Push a new scope and add the main method parameter.
    - Set the return type to `void`.
    - Visit the main method body.
- **Tip:** This method is similar to `visit_MethodDecl` but with rules specific to `main`.

`visit_ParamDecl` and `visit_ParamList`

- **Action:**
  Process method parameters.
- **Checks:**
    - Ensure that parameters are not declared more than once in the same scope.
        - **Semantic Error:** If a parameter is re-declared, raise `PARAMETER_ALREADY_DECLARED`.
    - Register each parameter's name and type in the symbol table.

- **Tip:** Use the type map to assign the correct type to each parameter.

`visit_Compound`

- **Action:**
  Process a block (compound statement).
- **Checks:**
  - For nested compound statements, push a new scope to handle local declarations.
  - Visit each statement within the block.
- **Tip:** Always pop the scope after processing the block.

`visit_If`

- **Action:**
  Process an if-statement.
- **Checks:**
  - Visit the condition expression and ensure its type is boolean.
    - **Semantic Error:** If the condition is not boolean, raise `CONDITIONAL_EXPRESSION_TYPE_MISMATCH`.
  - Push a new scope for the "then" branch and visit its statements.
  - If an "else" branch exists, push a new scope and visit it.
- **Tip:** Use `assert_semantic` to report type mismatches.

`visit_While` and `visit_For`

- **Action:**
  Process loop statements.
- **Checks:**
  - Save a reference to the current loop (for validating `break` statements).
  - Visit the loop condition and check that its type is boolean.
    - **Semantic Error:** If not boolean, raise `CONDITIONAL_EXPRESSION_TYPE_MISMATCH`.
  - For `For` loops, also visit the initialization and increment expressions.
  - Push a new scope for the loop body and process the contained statements.
  - Pop the scope and clear the loop reference after processing.
- **Tip:** Ensure that any `break` statement within the loop correctly identifies the enclosing loop. If a break is used outside any loop, raise `WRONG_BREAK_STATEMENT`.

`visit_Print`

- **Action:**
  Process a print statement.
- **Checks:**
  - Visit the expression(s) to be printed.
  - Ensure that each printed expression is of an allowed type (e.g., `char`, `int`, or `String`).
    - **Semantic Error:** If an expression is of an unsupported type, raise `PRINT_EXPRESSION_TYPE_MISMATCH`.
- **Tip:** Handle both single expressions and lists of expressions.

`visit_Assert`

- **Action:**
  Process an assert statement.
- **Checks:**
  - Visit the assert expression and verify that it is of boolean type.
    - **Semantic Error:** If not boolean, raise `ASSERT_EXPRESSION_TYPE_MISMATCH`.
- **Tip:** Ensure the condition of assert expressions is correctly validated.

`visit_Break`

- **Action:**
  Process a break statement.
- **Checks:**
  - Ensure that the break statement occurs within a loop.
    - **Semantic Error:** If not, raise `WRONG_BREAK_STATEMENT`.
- **Tip:** Use the stored loop reference to validate the break.

`visit_Return`

- **Action:**
  Process a return statement.
- **Checks:**
  - If an expression is provided, visit it to determine its type.
  - Compare the return expression's type against the current method's declared return type.
    - **Semantic Error:** If they do not match, raise `RETURN_TYPE_MISMATCH`.
- **Tip:** Use the current method context (set in `visit_MethodDecl` or `visit_MainMethodDecl`) to perform the comparison.

`visit_Assignment`

- **Action:**
  Process an assignment expression.
- **Checks:**
  - Visit both the left-hand side (lvalue) and right-hand side (rvalue).
  - Ensure that the identifier being assigned to is declared.
    - **Semantic Error:** If not declared, raise `UNDECLARED_NAME`.
  - Verify that the type of the rvalue is compatible with the type of the lvalue.
    - **Semantic Error:** If the types do not match, raise `ASSIGN_TYPE_MISMATCH`.
- **Tip:** Use `assert_semantic` to report type compatibility issues.

`visit_BinaryOp`

- **Action:**
  Process a binary operation.
- **Checks:**
  - Visit both the left and right operand expressions.
  - Verify that both operands have the same type.
    - **Semantic Error:** If the types differ, raise `BINARY_EXPRESSION_TYPE_MISMATCH`.
  - Check that the operator is supported by the operand type (using the type's `binary_ops and rel_ops`).
    - **Semantic Error:** If unsupported, raise `UNSUPPORTED_BINARY_OPERATION`.
  - Determine and assign the result type:
    - For arithmetic operators, the result is usually the same as the operand type.
    - For relational operators, the result is of type `boolean`.
- **Tip:** Leverage the operator sets from the type definitions.

`visit_UnaryOp`

- **Action:**
  Process a unary operation.
- **Checks:**
  - Visit the operand expression.
  - Ensure that the operator is supported by the operand's type (using the type's `unary_ops`).
    - **Semantic Error:** If not supported, raise `UNSUPPORTED_UNARY_OPERATION`.

    - Set the result type to be the same as the operand's type.
- **Tip:** Report unsupported operations with `assert_semantic`.

`visit_ArrayRef`

- **Action:**
  Process an array reference expression.
- **Checks:**
    - Visit the subscript expression and verify that its type is `int`.
        - **Semantic Error:** If not, raise `ARRAY_DIMENTION_MISMATCH`.
    - Visit the array name and confirm that it is of an array type (e.g., `IntArrayType` or `CharArrayType`).
        - **Semantic Error:** If the name's type is not an array type, raise `ARRAY_REF_TYPE_MISMATCH`.
    - Set the node's type to the array's element type.
- **Tip:** Use the array type's `element_type` attribute when setting the type.

`visit_FieldAccess`

- **Action:**
  Process field access on an object.
- **Checks:**
    - Visit the object expression.
    - Ensure that the object is defined and is of an object type (not a primitive).
        - **Semantic Error:** If the object is not defined, raise `UNDECLARED_NAME`; if it's a primitive, raise `OBJECT_TYPE_MUST_BE_A_CLASS`.
    - Check that the requested field exists in the object's class or its parent classes.
        - **Semantic Error:** If the field is not found, raise `UNDECLARED_FIELD`.
    - Set the resulting type based on the field's definition.
- **Tip:** Consider inheritance when merging fields from the class and its parent.

`visit_MethodCall`

- **Action:**
  Process a method invocation.
- **Checks:**
    - Visit the object on which the method is called.
    - Verify that the object is defined and is of an object type.
        - **Semantic Error:** If the object is not declared, raise `UNDECLARED_NAME`; if it's not an object, raise `OBJECT_TYPE_MUST_BE_A_CLASS`.
    - Check that the method exists in the object's class.
        - **Semantic Error:** If not, raise `UNDECLARED_METHOD`.
    - Visit the argument expressions and validate:
        - The number of arguments matches the method signature.
            - **Semantic Error:** If there is a mismatch, raise `ARGUMENT_COUNT_MISMATCH`.
        - Each argument's type matches the corresponding parameter type.
            - **Semantic Error:** If a type does not match, raise `PARAMETER_TYPE_MISMATCH`.
    - Set the node's type to the method's return type.
- **Tip:** Use the method signature stored in the class metadata for validations.

`visit_Length`

- **Action:**
  Process the length operator (applied to arrays or strings).
- **Checks:**

- Visit the expression whose length is requested.
    - Verify that the expression is either an array or a string.
        - **Semantic Error:** If not, raise `INVALID_LENGTH_TARGET`.
    - Set the type of the length expression to `int`.
- **Tip:** Ensure that the target is valid before assigning the type.

`visit_NewArray`

- **Action:**
  Process new array creation.
- **Checks:**
    - Visit the expression that determines the array size.
    - Confirm that the size expression is of type `int`.
        - **Semantic Error:** If not, raise `ARRAY_DIMENTION_MISMATCH`.
    - Create the corresponding array type instance (using `IntArrayType` or `CharArrayType`) and assign it to the node.
- **Tip:** Ensure that the element type is correctly used in the array type constructor.

`visit_NewObject`

- **Action:**
  Process new object creation.
- **Checks:**
    - Check that the class specified for the new object exists in the global symbol table.
        - **Semantic Error:** If the class is not found, raise `UNDECLARED_CLASS`.
    - Set the node's type to an `ObjectType` associated with the given class.
- **Tip:** Validate class existence before setting the type.

`visit_Constant`

- **Action:**
  Process a literal constant.
- **Checks:**
    - Directly assign the corresponding MJType (using the type map) to the constant node.
- **Tip:** Ensure the constant's type (e.g., `int`, `char`) matches the expected MJType.

`visit_This`

- **Action:**
  Process the `this` expression.
- **Checks:**
    - Set the node's type to the current class type.
    - Record the current class name.
- **Tip:** The current class context must be set in `visit_ClassDecl` before using `this`.

`visit_ID`

- **Action:**
  Process an identifier.
- **Checks:**
    - Look up the identifier in the symbol table stack.
    - Ensure that the identifier is declared.
        - **Semantic Error:** If not found, raise `UNDECLARED_NAME`.
    - Set the node's type and record the scope where it was declared.
- **Tip:** Report undeclared identifiers immediately.

`visit_Type`

- **Action:**
  Process a type specifier.
- **Checks:**
  - Use the type map to resolve the string name to the corresponding MJType.
- **Tip:** This is a straightforward visit to support other nodes.

`visit_Extends`

- **Action:**
  Process inheritance.
- **Checks:**
  - Verify that the parent (super) class exists in the global symbol table.
    - **Semantic Error:** If the parent class is undeclared, raise `UNDECLARED_CLASS`.
- **Tip:** Check the existence of the super class early in class processing.

`visit_ExprList` and `visit_InitList`

- **Action:**
  Process lists of expressions.
- **Checks for `ExprList`:**
  - Visit each expression in the list.
- **Checks for `InitList`:**
  - Visit each initializer expression.
  - For `InitList`, ensure every expression is a constant.
    - **Semantic Error:** If an initializer is not a constant, raise `NOT_A_CONSTANT`.
  - Set the node’s type to the correct array type based on the elements.
- **Tip:** Handle the possibility of an empty list appropriately if needed.