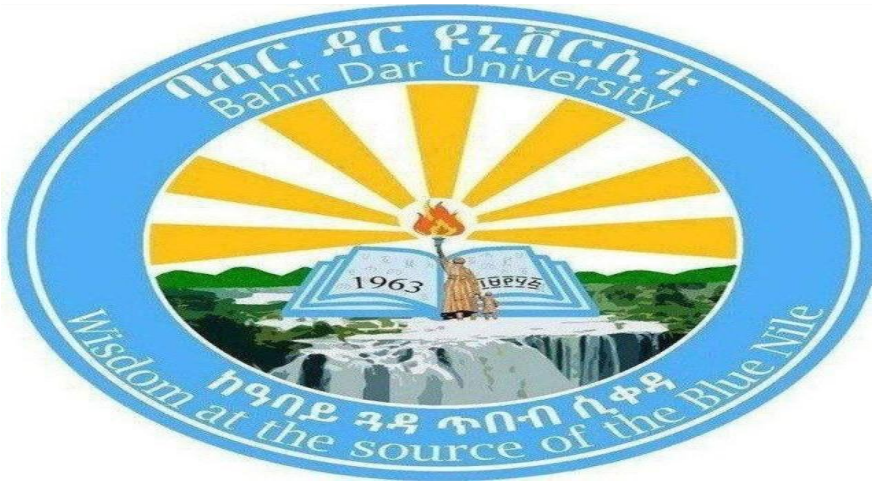

BAHIRDAR INSTUETE OF TECHNOLOGY

FACULITY OF COMPUTING

DEPARTMENT OF SOFTWARE ENGINEERING

PRINCIPLE OF COMPILER DESIGN

INDIVUDUAL PROJECT

ERMIYAS MISGANAW……..1505942

QUESTION -70

Enforce semantic correctness for static variables in classes.

You must validate correct usage of static variables, ensuring proper initialization rules, scope handling, and restrictions on access where applicable.

ENFORCING SEMANTIC CORRECTNESS FOR STATIC VARIABLES IN CLASSES

enforcing semantic correctness for **static variables in classes** involves verifying several key properties during compilation.

1. Declaration Before Use

- **Rule**: Static variables must be declared before they are referenced.
- **Mechanism**:
  - As shown in the document (pages 23–24), inherited attributes (e.g., L.inh) can propagate type information down declaration lists.
  - The symbol table (page 8) is used to record variable declarations. Before a static variable is used, the compiler must check its entry in the symbol table.
- **Example SDD Rule**:

  text

  Production: ClassBody → StaticDecl MemberList
  Semantic: Insert(StaticDecl.entry) into symbol table

2. Initialization Rules

- **Rule**: Static variables must be initialized according to language-specific rules (e.g., only constants, or allowed runtime expressions).
- **Mechanism**:

- Use **synthesized attributes** (page 18) to compute and check the initialization value.
- For compile-time constants, the attribute can store the value; otherwise, the compiler must ensure the expression is valid in a static context.

- **Example**:

  text

  Production: StaticDecl → static T id = Expr ;
  Semantic: if Expr.type != T.type → Error "Type mismatch"
  if Expr.isConstant == false → Error "Must be constant"

3. Scope and Visibility

- **Rule**: Static variables belong to the class, not instances. They should be accessed via the class name, not through object references.
- **Mechanism**:
  - Inherited attributes can track the current context (e.g., insideStaticMethod flag).
  - The symbol table entry for a static variable includes a isStatic flag.
- **Example Check**:

  text

  On reference `x.y`:
  if y is static and x is an object instance →
  Warning/Error: "Static member accessed via instance"

4. Access Restrictions

- **Rule**: Static variables may have modifiers (private, protected, public) that restrict access.

- **Mechanism**:
  - The symbol table stores visibility attributes.
  - During access, the compiler checks if the current scope is permitted (e.g., a private static variable is only accessible within the same class).
- **Example SDD Rule**:

  text

  ```
  Production:   Access → ClassName . staticVar
  Semantic:     if not visible(staticVar, currentClass) →
                    Error "Cannot access private static variable"
  ```

5. Single Initialization Enforcement

- **Rule**: Static variables should not be reinitialized in a way that violates language semantics (e.g., in Java, static finals can only be assigned once).
- **Mechanism**:
  - Use a **boolean attribute** initialized in the symbol table.
  - On assignment, check if initialized == true and variable is final → error.
- **Example**:

  text

  ```
  On assignment to static final variable:
    if symbolTable[var].initialized →
      Error "Cannot reassign final static variable"
  ```

6. Type Compatibility in Assignments and Operations

- **Rule**: Any assignment or operation involving static variables must respect type rules.
- **Mechanism**:

- As shown in the **attribute grammar example** (pages 15–16), synthesized attributes like val and type are computed and compared.
- For static variables, the type attribute is retrieved from the symbol table and checked against the expression type.

- **Example**:

  text

  Production: Expr → staticVar + Expr2
  Semantic: if staticVar.type != Expr2.type →
  Error "Type mismatch in static variable operation"

7. Lifetime and Memory Consistency

- **Rule**: Static variables exist for the duration of the program. Semantic analysis must ensure they are not used in contexts that assume instance-specific lifetime (e.g., in destructors or instance-specific clean-up).
- **Mechanism**:
  - Inherited attributes can track whether the current context is static or instance-based.
  - If a static variable is referenced in a non-static cleanup block → issue warning.

Summary of Semantic Checks for Static Variables

Check	Mechanism	Document Reference
Declaration before use	Symbol table lookup	Pages 8, 23–24
Proper initialization	Synthesized attributes for constant evaluation	Pages 18–20
Scope correctness	Inherited context attribute	Pages 21–24
Access modifiers	Symbol table visibility flag	Page 8
Single assignment	Boolean attribute in symbol table	Page 32
Type compatibility	Type attribute comparison	Pages 15–16
Lifetime consistency	Context tracking via inherited attributes	Pages 37–38

➲ By integrating these checks into the **semantic analysis phase**, the compiler can enforce correct usage of static variables, ensuring they are **well-defined, well-typed, and well-scoped**, as required by the language specification.