BARZ Compiler: An Overview of Symbol Table Generation and Semantic Analysis

Ismael Talib Ridha Barzani
Student ID: 40188139
Department of Electrical and Computer Engineering
Concordia University - Gina Cody School
Montreal, Quebec, Canada
ismaelmergasori@gmail.com

***Abstract*—Symbol table generation and semantic analysis are critical components in a compiler's front-end that bridge syntactic structures to meaningful program validation. Following the AST generation phase, these components enable the compiler to detect semantic errors that cannot be identified through syntax analysis alone. The symbol table provides a structured repository of all program identifiers along with their associated attributes such as type, scope, and visibility, facilitating name resolution and type checking. In this report, I present the design and implementation of the semantic analyzer component for the BARZ compiler. First, I describe the hierarchical symbol table structure that efficiently captures nested scopes, inheritance relationships, and function signatures. Second, I detail the semantic checking visitor that traverses the AST to verify type compatibility, variable declarations, scope rules, and inheritance constraints. The implementation employs a two-phase approach: an initial pass builds the symbol table while capturing declaration-related errors, followed by a thorough semantic analysis that identifies type mismatches, undeclared identifiers, duplicate identifiers, and improper function calls. The system successfully detects a wide range of errors including overridden members, visibility violations, and array dimension mismatches, demonstrating how semantic analysis complements syntactic parsing to ensure program correctness.**

***Index Terms*—Compiler Design, Symbol Table, Semantic Analysis, Type Checking**

I. Introduction

The parsing phase of a compiler establishes syntactic correctness but cannot verify that a program adheres to semantic rules such as type compatibility and proper identifier usage. Symbol table generation and semantic analysis bridge this gap, verifying that syntactically correct programs also follow the language's semantic constraints. The symbol table serves as a centralized hierarchical table structure for program entities, that captures identifiers, parameters, variables, types, scopes, and relationships between program elements. In the BARZ compiler, the hierarchical symbol table design accommodates nested scopes, inheritance relationships, and multi-dimensional array types while providing constant-time-complexity lookup mechanisms essential for semantic verification.

This is submitted as part of COMP-6421 Assignment #4.

Symbol table construction occurs through an initial traversal of the AST, recording declarations of classes, functions, and variables along with their attributes. This phase detects fundamental errors such as duplicate declarations, undefined references, and circular dependencies. Following this foundation, semantic analysis performs comprehensive validation including type checking, array bounds verification, visibility constraints, and proper function invocation. The implementation employs the visitor design pattern, allowing separate passes for symbol table construction and semantic validation while maintaining a uniform AST traversal mechanism.

The semantic analyzer for BARZ handles a diverse range of checks including class hierarchy verification, function overloading validation, parameter type matching, and proper use of object members. When violations are detected, the system produces detailed error messages with exact line location information, facilitating rapid debugging. This two-phase approach—symbol table construction followed by semantic validation—provides a robust framework that can be extended to accommodate additional language features. By focusing on both symbol management and semantic rule enforcement, this phase establishes the groundwork for subsequent compiler stages including code generation and optimization.

II. Semantic Rules Implementation

The semantic analyzer for the Barz language implements a comprehensive set of semantic validation rules to ensure program correctness beyond syntactic verification. These rules are organized into two phases: symbol table creation and semantic checking. Below is a checklist detailing the implementation status of each semantic rule, followed by relevant code references.

A. Symbol Table Creation Phase

✓ 1. Global scope table creation: A new table is created at the beginning of the AST traversal for

the global scope, serving as the root of the symbol table hierarchy.

- ✓ 2. Class declaration entries: Each class declaration creates an entry in the global table with links to local tables that contain the class members and methods.
- ✓ 3. Variable definition entries: All variable definitions (class data members, function parameters, and local variables) are recorded in their appropriate scope tables with complete type information including array dimensions.
- ✓ 4. Function definition entries: Function definitions create entries in the appropriate scope table with links to local tables containing parameter and local variable information.
- ✓ 5. Early error detection: During symbol table creation, the visitor detects and reports errors such as multiple declared identifiers in the same scope and issues warnings for overriden inherited members.
- ✓ 6. Member function declaration-definition correspondence: The system verifies that all declared member functions have a corresponding implementation and vice versa, reporting errors for undeclared or undefined functions.
- ✓ 7. Symbol table output: The complete symbol table hierarchy is written to an output file (.outsymboltables) displaying all scopes, classes, functions, and variables with their associated attributes.
- ✓ 8. Duplicate identifier detection: The analyzer prevents multiple declarations of the same identifier within a single scope, generating appropriate error messages for duplicate classes, data members, functions, or local variables.
- ✓ 9. Function overloading: Function overloading is supported for both member and free functions, with overloaded functions correctly distinguished by their parameter lists.

B. Semantic Checking Phase

- ✓ 10. Type checking: Expressions are analyzed to infer types, with type compatibility verified for assignments, return statements, and operators to ensure type safety throughout the program.
- ✓ 11. Identifier definition verification: All identifier references are validated against the symbol table to ensure they are defined in the appropriate scope, with clear error messages for undeclared variables, functions, or classes.
- ✓ 12. Function call validation: Function calls are verified to have the correct number and type of parameters, with proper type checking for all argument expressions against the declared parameter types.
- ✓ 13. Array access verification: Array variables are checked to ensure accesses use the correct dimensionality, all indices are of integer type, and array parameters have compatible dimensions with their corresponding formal parameter declarations.
- ✓ 14. Circular class dependency detection: The analyzer identifies and reports circular dependencies in class hierarchies (through inheritance or member composition) as semantic errors.
- ✓ 15. Dot operator validation: The member access operator (".") is verified to be used only on class type variables, with validation that the accessed member actually exists in the class, reporting appropriate errors for undeclared members.

C. Implementation Details

The following code extracts demonstrate how each semantic rule is implemented within the compiler.

1) Symbol Table Creation Phase Implementation:

a) 1. Global Scope Table Creation:

```
// In SymbolTableVisitor constructor
SymbolTableVisitor::SymbolTableVisitor() {
  globalTable =
  ↪  std::make_shared<SymbolTable>("global");
  currentTable = globalTable;
}

// In visitProgram method
void SymbolTableVisitor::visitProgram(ASTNode*
↪  node) {
  // Start at global scope
  currentTable = globalTable;
  // ...
}
```

b) 2. Class Declaration Entries:

```
// In visitClass method
void SymbolTableVisitor::visitClass(ASTNode*
↪  node) {
  // Get class name
  currentClassName =
  ↪  classIdNode->getNodeValue();

  // Create class symbol and table
  auto classSymbol = std::make_shared<Symbol>(
  ↪  currentClassName, currentClassName,
  ↪  SymbolKind::CLASS);
  globalTable->addSymbol(classSymbol);

  auto classTable =
  ↪  std::make_shared<SymbolTable>(
  ↪  currentClassName, globalTable.get());
  globalTable->addNestedTable(
  ↪  currentClassName, classTable);
}
```

c) 3. Variable Definition Entries:

```
// In visitVariable method - creates entries
↪  for data members
void
↪  SymbolTableVisitor::visitVariable(ASTNode*
↪  node) {
```

```
  // Create variable symbol with appropriate
  ↪ type and dimensions
  auto varSymbol =
  ↪ std::make_shared<Symbol>(varName,
  ↪ currentType, SymbolKind::VARIABLE);

  // Add array dimensions if any
  for (int dim : currentArrayDimensions) {
    varSymbol->addArrayDimension(dim);
  }

  // Set visibility and add to current table
  varSymbol->setVisibility(currentVisibility);
  currentTable->addSymbol(varSymbol);
}
```

d) 4. Function Definition Entries:

```
// In SemanticCheckingVisitor::visitFunction
void SemanticCheckingVisitor::visitFunction(
↪ ASTNode* node) {
  // Process function signature
  ASTNode* signatureNode =
  ↪ node->getLeftMostChild();
  if (signatureNode) {
    signatureNode->accept(this);

    // Create or get the function's symbol
    ↪ table
    std::string funcTableName =
    ↪ currentClassName.empty() ?
        "::" + currentFunctionName :
        ↪ currentClassName + "::" +
        ↪ currentFunctionName;
    auto functionTable =
    ↪ currentTable->getNestedTable(
    ↪ currentFunctionName);

    if (functionTable) {
      // Process function body in function's
      ↪ scope
      auto savedTable = currentTable;
      currentTable = functionTable;

      // Process function body
      ASTNode* bodyNode =
      ↪ signatureNode->getRightSibling();
      if (bodyNode) {
        bodyNode->accept(this);
      }

      // Restore table
      currentTable = savedTable;
    }
  }
}
```

e) 5. Early Error Detection:

```
// In checkOverridenInheritedMembers
void SemanticCheckingVisitor::
↪ checkOverridenInheritedMembers(const
↪ std::string& className) {
  // Get class and parent tables
  auto classSymbol =
  ↪ globalTable->lookupSymbol(className);
  auto classTable =
  ↪ globalTable->getNestedTable(className);
  std::string parentClass =
  ↪ inheritedClasses[0];
  auto parentTable =
  ↪ globalTable->getNestedTable(parentClass);

  // Check for overriden members
  for (const auto& [memberName, memberSymbol]
  ↪ : classTable->getSymbols()) {
    if (memberSymbol->getKind() ==
    ↪ SymbolKind::VARIABLE) {
      auto parentMember =
      ↪ parentTable->lookupSymbol(memberName);
      if (parentMember &&
      ↪ parentMember->getKind() ==
      ↪ SymbolKind::VARIABLE) {
        reportError("Data member '" +
        ↪ memberName + "' in class " +
        ↪ className +
                    " overides inherited member
                    ↪ from class " +
                    ↪ parentClass, nullptr);
      }
    }
  }
}
```

f) 6. Member Function Declaration/Definition Correspondence:

```
// In checkUndefinedMemberFunctions and
↪ checkUndeclaredMemberFunctions
void SemanticCheckingVisitor::
↪ checkUndefinedMemberFunctions() {
  // For each class, check if all declared
  ↪ functions are implemented
  for (const auto& [className, declared] :
  ↪ declaredFunctions) {
    auto implemented =
    ↪ implementedFunctions[className];

    for (const auto& funcName : declared) {
      if (implemented.find(funcName) ==
      ↪ implemented.end()) {
        reportError("Undefined member
        ↪ function: " + funcName + " in
        ↪ class " + className, nullptr);
      }
    }
  }
}

void SemanticCheckingVisitor::
↪ checkUndeclaredMemberFunctions() {
  // For each class, check if all implemented
  ↪ functions are declared
  for (const auto& [className, implemented] :
  ↪ implementedFunctions) {
    auto declared =
    ↪ declaredFunctions[className];

    for (const auto& funcName : implemented) {
      if (declared.find(funcName) ==
      ↪ declared.end()) {
        reportError("Undeclared member
        ↪ function definition: " + funcName
        ↪ +
                    " in class " + className,
                    ↪ nullptr);
      }
    }
  }
}
```

g) 7. Symbol Table Output:

```
// In outputSymbolTable
void
↪  SymbolTableVisitor::outputSymbolTable(const
↪  std::string& filename) {
  // Output symbol table to file
  std::ofstream outFile(filename);
  writeTableToFile(outFile, globalTable);
  outFile.close();
}
```

h) 8. Duplicate Identifier Detection:

```
// In checkDuplicateClassDeclarations
void SemanticCheckingVisitor::
↪  checkDuplicateClassDeclarations() {
  std::unordered_map<std::string, int>
  ↪  classCount;

  // Count occurrences of each class name
  for (const auto& [className, table] :
  ↪  globalTable->getNestedTables()) {
    auto classSymbol =
    ↪  globalTable->lookupSymbol(className);
    if (classSymbol && classSymbol->getKind()
    ↪  == SymbolKind::CLASS) {
      classCount[className]++;
    }
  }

  // Report duplicate class declarations
  for (const auto& [className, count] :
  ↪  classCount) {
    if (count > 1) {
      reportError("Multiple declared class: "
      ↪  + className, nullptr);
    }
  }
}
```

i) 9. Function Overloading:

```
// In checkDuplicateFunctionDeclarations
void SemanticCheckingVisitor::
↪  checkDuplicateFunctionDeclarations() {
  std::unordered_map<std::string,
  ↪  std::vector<std::shared_ptr<Symbol>>>
  ↪  functionSymbols;

  // Collect function symbols with same name
  for (const auto& [funcName, symbol] :
  ↪  globalTable->getSymbols()) {
    if (symbol->getKind() ==
    ↪  SymbolKind::FUNCTION) {
      functionSymbols[funcName].push_back(
      ↪  symbol);
    }
  }

  // Check for functions with same name and
  ↪  signature
  for (const auto& [funcName, symbols] :
  ↪  functionSymbols) {
    for (size_t i = 0; i < symbols.size();
    ↪  i++) {
      for (size_t j = i + 1; j <
      ↪  symbols.size(); j++) {
        // Compare parameter lists to
        ↪  distinguish overloaded functions
        const auto& params1 =
        ↪  symbols[i]->getParams();
        const auto& params2 =
        ↪  symbols[j]->getParams();

        // If same name, same param count, and
        ↪  same param types - it's a
        ↪  duplicate
        if (params1.size() == params2.size())
        ↪  {
          bool allSame = true;
          for (size_t k = 0; k <
          ↪  params1.size(); k++) {
            if (params1[k] != params2[k]) {
              allSame = false;
              break;
            }
          }

          if (allSame && symbols[i]->getType()
          ↪  == symbols[j]->getType()) {
            reportError("Multiple defined free
            ↪  function: " + funcName,
            ↪  nullptr);
          }
        }
      }
    }
  }
}
```

2) Semantic Checking Phase Implementation:

a) 10. Type Checking:

```
// In visitAssignment
void SemanticCheckingVisitor::visitAssignment(
↪  ASTNode* node) {
  // Process left and right sides
  ASTNode* leftNode =
  ↪  node->getLeftMostChild();
  if (leftNode) {
    leftNode->accept(this);
    TypeInfo leftType = currentExprType;

    // Process right side
    ASTNode* rightNode =
    ↪  opNode->getRightSibling();
    if (rightNode) {
      rightNode->accept(this);
      TypeInfo rightType = currentExprType;

      // Check type compatibility
      if (!areTypesCompatible(leftType,
      ↪  rightType)) {
        reportError("Type mismatch in
        ↪  assignment. Left side is " +
        ↪  formatTypeInfo(leftType) +
                    " but right side is " +
                    ↪  formatTypeInfo(
                    ↪  rightType), node);
      }
    }
  }
}

// In visitReturnStatement
void SemanticCheckingVisitor::
↪  visitReturnStatement( ASTNode* node) {
  // Check return type against function's
  ↪  declared return type
  ASTNode* exprNode =
  ↪  node->getLeftMostChild();
  if (exprNode) {
    exprNode->accept(this);
    TypeInfo returnType = currentExprType;
```

```
    // Check type compatibility
    TypeInfo expectedType;
    expectedType.type =
    ↪  expectedReturnType.back();

    if (!areTypesCompatible(expectedType,
    ↪  returnType)) {
      reportError("Return type mismatch.
      ↪  Expected " + expectedType.type +
                  ", got " + returnType.type,
                  ↪  node);
    }
  }
}
```

b) 11. Identifier Definition Verification:

```
// In visitIdentifier
void SemanticCheckingVisitor::visitIdentifier(
↪  ASTNode* node) {
  std::string idName = node->getNodeValue();

  // Look for identifier in current scope and
  ↪  parent scopes
  auto symbol =
  ↪  currentTable->lookupSymbol(idName);
  if (!symbol) {
    reportError("Use of undeclared variable: "
    ↪  + idName, node);
    currentExprType.type = "error";
    currentExprType.dimensions.clear();
    currentExprType.isClassType = false;
    return;
  }

  // Set type information
  currentExprType.type = symbol->getType();
  currentExprType.dimensions =
  ↪  symbol->getArrayDimensions();
  currentExprType.isClassType =
  ↪  globalTable->lookupSymbol(
  ↪  currentExprType.type) != nullptr;
}
```

c) 12. Function Call Validation:

```
// In visitFunctionCall
void
↪  SemanticCheckingVisitor::visitFunctionCall(
↪  ASTNode* node) {
  // Get function name and gather parameter
  ↪  types
  std::string funcName =
  ↪  funcIdNode->getNodeValue();

  // Gather parameter types
  std::vector<TypeInfo> paramTypes;
  // [Parameter collection code]

  // Check parameter count
  const auto& declaredParams =
  ↪  funcSymbol->getParams();
  if (declaredParams.size() !=
  ↪  paramTypes.size()) {
    reportError("Function " + funcName + "
    ↪  called with wrong number of arguments.
    ↪  Expected " +
                std::to_string(
                ↪  declaredParams.size()) + ",
                ↪  got " +
                std::to_string(
                ↪  paramTypes.size()), node);
  } else {
    // Check each parameter type
    for (size_t i = 0; i <
    ↪  declaredParams.size(); ++i) {
      TypeInfo declaredType =
      ↪  parseTypeString(declaredParams[i]);

      if (!areTypesCompatible(declaredType,
      ↪  paramTypes[i])) {
        reportError("Function " + funcName + "
        ↪  parameter " + std::to_string(i+1)
        ↪  +
                    " type mismatch. Expected "
                    ↪  + formatTypeInfo(
                    ↪  declaredType) +
                    ", got " + formatTypeInfo(
                    ↪  paramTypes[i]), node);
      }
    }
  }
}
```

d) 13. Array Access Verification:

```
// In visitArrayAccess
void
↪  SemanticCheckingVisitor::visitArrayAccess(
↪  ASTNode* node) {
  // Check array dimensions and index types
  ASTNode* arrayNode =
  ↪  node->getLeftMostChild();
  if (arrayNode) {
    arrayNode->accept(this);
    TypeInfo arrayType = currentExprType;

    // Check if this is actually an array
    if (arrayType.dimensions.empty()) {
      reportError("Array index used on
      ↪  non-array type: " + arrayType.type,
      ↪  node);
      currentExprType.type = "error";
      return;
    }

    // Count indices and check their types
    int indexCount = 0;
    ASTNode* indexNode =
    ↪  arrayNode->getRightSibling();
    while (indexNode) {
      indexCount++;
      indexNode->accept(this);
      TypeInfo indexType = currentExprType;

      // Index must be an integer
      if (indexType.type != "int") {
        reportError("Array index must be of
        ↪  integer type, got: " +
        ↪  indexType.type, node);
      }

      indexNode =
      ↪  indexNode->getRightSibling();
    }

    // Check dimension count
    if (indexCount !=
    ↪  arrayType.dimensions.size()) {
      reportError("Use of array with wrong
      ↪  number of dimensions. Expected " +
                  std::to_string(
                  ↪  arrayType.dimensions.size())
                  ↪  + ", got " +
```

```
                    std::to_string(indexCount),
                    ↪  node);
    }
  }
}
```

e) 14. Circular Class Dependency Detection:

```
// In checkClassCircularDependency
bool SemanticCheckingVisitor::
↪  checkClassCircularDependency(const
↪  std::string& className,
      std::unordered_set<std::string>&
      ↪  visited) {
  // If we've already visited this class in
  ↪  this path, we have a cycle
  if (visited.find(className) !=
  ↪  visited.end()) {
    return true;
  }

  // Mark this class as visited
  visited.insert(className);

  // Check for circular inheritance
  const auto& inheritedClasses =
  ↪  classSymbol->getInheritedClasses();
  for (const auto& parentClass :
  ↪  inheritedClasses) {
    if (checkClassCircularDependency(
    ↪  parentClass, visited)) {
      return true;
    }
  }

  // Check for circular dependencies through
  ↪  data members
  for (const auto& [memberName, memberSymbol]
  ↪  : classTable->getSymbols()) {
    if (memberSymbol->getKind() ==
    ↪  SymbolKind::VARIABLE) {
      std::string memberType =
      ↪  memberSymbol->getType();
      // If the member is of a class type,
      ↪  check for circular dependency
      auto memberTypeSymbol =
      ↪  globalTable->lookupSymbol(
      ↪  memberType);
      if (memberTypeSymbol &&
      ↪  memberTypeSymbol->getKind() ==
      ↪  SymbolKind::CLASS) {
        if (checkClassCircularDependency(
        ↪  memberType, visited)) {
          return true;
        }
      }
    }
  }

  // Remove this class from the visited set
  ↪  since we're done exploring this path
  visited.erase(className);

  return false; // No circular dependency
  ↪  found
}
```

f) 15. Dot Operator Validation:

```
// In visitDotIdentifier
void SemanticCheckingVisitor::
↪  visitDotIdentifier(ASTNode* node) {
  // Process object
  ASTNode* objNode = node->getLeftMostChild();
  if (objNode) {
    objNode->accept(this);
    TypeInfo objType = currentExprType;

    // Check if this is a class type
    if (!objType.isClassType) {
      reportError("Dot operator used on
      ↪  non-class type: " + objType.type,
      ↪  node);
      currentExprType.type = "error";
      return;
    }

    // Process member and check its existence
    ASTNode* memberNode =
    ↪  objNode->getRightSibling();
    if (memberNode) {
      std::string memberName =
      ↪  memberNode->getNodeValue();

      // Look up member in class
      auto classTable =
      ↪  globalTable->getNestedTable(
      ↪  objType.type);
      auto memberSymbol =
      ↪  classTable->lookupSymbol(memberName);
      if (!memberSymbol) {
        reportError("Undeclared member: " +
        ↪  memberName + " in class " +
        ↪  objType.type, node);
        currentExprType.type = "error";
        return;
      }

      // Check visibility - only allow access
      ↪  to public members from outside
      if (memberSymbol->getVisibility() ==
      ↪  Visibility::PRIVATE &&
      ↪  currentClassName != objType.type) {
        reportError("Cannot access private
        ↪  member: " + memberName + " in
        ↪  class " + objType.type, node);
        currentExprType.type = "error";
        return;
      }
    }
  }
}
```

This implementation demonstrates a comprehensive approach to semantic analysis, with relevant error detection and reporting capabilities. The two-phase design (symbol table construction followed by semantic checking) provides a solid foundation for ensuring program correctness beyond syntactic validity. By implementing all 15 semantic rules, the BARZ compiler can detect a wide range of potential programming errors before execution, significantly improving code reliability.

III. Design

A. Overall Design

The semantic analyzer for the compiler employs a modular design centered around the visitor pattern, which facilitates orderly traversal of the Abstract Syntax

Tree (AST) while performing different semantic operations. The visitor pattern is possible as the compiler is written in C++. This architecture was chosen to maintain a clean separation of dependencies and to enable multiple passes over the AST without duplicating traversal logic.

1) Component Architecture: The solution consists of several interrelated components:

- **Visitor Base Class**: Provides the foundation for all AST traversal operations through a generic visitor interface, allowing specialized visitor implementations to define specific behaviors for each AST node type.
- **Symbol Table Hierarchy**: A multi-level structure that models nested scopes in the program, with the global scope at the root and nested tables for classes, functions, and blocks. This hierarchical design mirrors the natural scope nesting in the language.
- **Symbol Class**: Represents program entities (classes, functions, variables) with attributes for type, visibility, parameters, and array dimensions. These rich entities support complex semantic checks involving inheritance, accessibility, and type compatibility.
- **SymbolTableVisitor**: Specialized visitor responsible for constructing the symbol table hierarchy during the first traversal pass, capturing all declarations and their properties.
- **SemanticCheckingVisitor**: Specialized visitor that performs detailed semantic verification during the second traversal pass, using the previously constructed symbol tables to validate program semantics.
- **Type System**: A type representation and compatibility checking system that supports primitive types, arrays, and class types with inheritance relationships.

2) Design Rationale: The two-visitor approach was selected for several compelling reasons:

1) **Separation of Concerns**: By dividing symbol collection and semantic checking into distinct phases, each visitor maintains a clear, focused responsibility.
2) **Dependency Management**: Semantic checks depend on a complete symbol table. The two-phase approach ensures all declarations are processed before any semantic verification begins.
3) **Error Handling Clarity**: Declaration errors are detected and reported separately from semantic errors, providing clearer feedback to programmers about the nature of their mistakes.
4) **Complexity Management**: Breaking the semantic analysis into two phases reduces the complexity of each visitor class, making the code more maintainable and extensible.
5) **Forward Declaration Support**: The two-phase approach naturally accommodates forward declarations and mutual recursion in the source language.

The symbol table's hierarchical design directly models the nested scoping rules of the compiler. This design choice facilitates efficient name resolution by allowing the system to search progressively outward from the current scope. The shared symbol table between visitors ensures that semantic analysis has complete access to all declaration information gathered during the first phase.

B. Phases

The semantic analysis process is divided into two distinct phases, each serving a specific purpose in ensuring program correctness.

1) Phase 1: Symbol Table Construction: The first phase is responsible for traversing the AST to identify and record all declarations in the program, building a comprehensive symbol table hierarchy. This phase:

- **Establishes Scope Hierarchy**: Creates nested tables for global scope, classes, and functions, establishing the containment relationships between scopes.
- **Records Declarations**: Captures all identifier declarations, including their names, types, visibility, and other attributes.
- **Processes Inheritance**: Records inheritance relationships between classes for later validation.
- **Collects Function Signatures**: Records function declarations with their parameter types and return types for later verification of calls.
- **Validates Declarations**: Performs immediate validation of declarations, detecting duplicate declarations and other declaration-level issues.
- **Maps Array Dimensions**: Records array dimension information for later bounds checking.
- **Handles Member Visibility**: Records private/public designations for class members.

This phase is essential because it establishes the foundational information needed for all subsequent semantic checks. By completing the symbol table before beginning semantic verification, we ensure that forward references and mutual dependencies between program elements are properly resolved.

2) Phase 2: Semantic Checking: The second phase leverages the completed symbol table to perform comprehensive semantic validation, ensuring that the program follows all semantic rules of the Barz language. This phase:

- **Performs Type Checking**: Verifies type compatibility in expressions, assignments, and function calls.
- **Validates References**: Ensures all identifiers used in the program are properly declared before use.
- **Checks Function Calls**: Verifies that function calls match their declarations in parameter count and types.
- **Validates Array Access**: Ensures array accesses use the correct number of dimensions and integer indices.
- **Enforces Visibility Rules**: Checks that private members are only accessed from within their containing class.
- **Detects Circular Dependencies**: Identifies and reports circular class dependencies that would cause logical issues.
- **Ensures Type Safety**: Verifies that operations are applied to compatible types and that conversions are valid.
- **Validates Inheritance**: Ensures proper member overriding and inheritance relationships.

The second phase is critical because it identifies semantic errors that would lead to incorrect program behavior despite syntactically correct code. By verifying that the program follows all semantic constraints of the language, it ensures that the code is not just structurally valid but logically coherent and type-safe.

3) Phase Integration: While conceptually distinct, these two phases are integrated within the compilation pipeline:

1) The AST is first processed by the SymbolTableVisitor, which constructs the complete symbol table hierarchy.
2) Any errors detected during symbol table construction are captured for reporting.
3) The SemanticCheckingVisitor is then initialized with the completed symbol table.
4) The AST is traversed a second time by the SemanticCheckingVisitor, performing all semantic validations.
5) Errors from both phases are consolidated and reported with source location information.

This two-phase approach provides a robust foundation for semantic analysis, ensuring that all declarations are properly recorded before verification begins, while maintaining a clear separation of concerns between symbol collection and semantic validation.

IV. Tools

The implementation of the semantic analyzer relies on a selected set of tools, libraries, and programming techniques. The choices made reflect a balance between performance requirements, maintainability, and the specific needs of a compiler's semantic analysis phase.

A. Core Technologies

- **C++ Programming Language**: C++ was selected for its performance characteristics and robust support for systems programming. Its strong typing system and compile-time checks align well with implementing a compiler that performs similar tasks. The ability to use both object-oriented and procedural paradigms provided necessary flexibility for different aspects of the implementation, specifically the ability to support the visitor pattern.
- **C++ Standard Library**: The standard library provided essential data structures and algorithms without introducing external dependencies. Specifically:
  - `std::unordered_map` for efficient symbol lookup (O(1) average complexity)
  - `std::vector` for ordered collections like array dimensions and parameter lists
  - `std::shared_ptr` for safe, reference-counted memory management
  - `std::ofstream` for structured file output of symbol tables and error messages
  - `std::algorithms` for reversing the arrays and vectors
- **Visitor Design Pattern**: This pattern was crucial for separating traversal logic from semantic operations, allowing clean extension of functionality without modifying the AST structure. It enables multiple traversal strategies (symbol collection and checking) over the same tree structure.

B. Design Techniques

- **Smart Pointers**: Used throughout the implementation to manage memory automatically, preventing leaks while maintaining clear ownership semantics. Smart pointers were chosen over manual memory management to eliminate a common source of bugs in compiler implementations.
- **Two-Phase Design**: The separation into symbol table construction and semantic checking phases was a deliberate design technique that simplified the implementation and improved error reporting.
- **Type System Abstraction**: The `TypeInfo` structure abstracts type representation, making the analyzer extensible to additional types without extensive refactoring.

The selected tools and techniques prioritize robustness, maintainability, and performance—essential qualities for a compiler component that must handle complex language constructs while producing precise error messages.