BAHIR DAR UNIVERSITY

BAHIR DAR INSTITUTE OF TECHNOLOGY

COMPUTING FACULTY

DEPARTMENT OF COMPUTER SCIENCE

Course Title: Compiler Design(CoSc4022)

Individual Assignment

Topic: Syntax Analysis

Student Name: Adem Seidu

ID Number:1411142

Submitted to:Mr.Wendimu B

Submission date: 01/05/2015EC

1. Introduction

Semantic analysis is a critical phase of compiler design that ensures a program is **meaningfully correct** beyond syntactic structure. One of its core responsibilities is **scope management**, implemented using **symbol tables**.
This assignment focuses on extending symbol table logic to **correctly handle variable shadowing**, while also **detecting and warning about potentially harmful shadowing practices** that may lead to subtle program errors.

---

2. Variable Shadowing

2.1 Formal Definition

In compiler theory, **variable shadowing** occurs when an identifier declared in an **inner scope** has the same name as an identifier declared in an **outer scope**, causing the outer declaration to become inaccessible within the inner scope.

Formally:

Given two declarations d1 and d2 of identifier x, where scope(d2) is nested inside scope(d1), declaration d2 shadows d1 within its scope.

2.2 Legal vs. Harmful Shadowing

Type	Description
Legal Shadowing	Shadowing allowed by the language rules, typically in inner scopes
Harmful Shadowing	Shadowing that reduces code clarity or causes unintended behavior
Illegal Shadowing	Redeclaration of an identifier within the same scope

Examples of **harmful shadowing** include:

- Shadowing function parameters
- Shadowing global variables
- Shadowing class fields inside methods

---

3. Scope and Symbol Table Design

3.1 Hierarchical Symbol Table Structure

The compiler maintains a **hierarchical (stack-based) symbol table**, where each scope has its own table.

Typical scopes include:

1. **Global Scope**
2. **Function Scope**
3. **Block Scope** (e.g., {} in loops or conditionals)
4. **Class Scope** (if object-oriented features exist)

Each symbol table contains:

- Identifier name
- Type information
- Scope level
- Additional attributes (parameter, field, variable)

3.2 Scope Creation and Destruction

- A **new scope** is created when entering:
    - A function
    - A block
    - A class definition
- The scope is **destroyed** when exiting that construct.

Scopes are typically managed using a **stack**.

3.3 Identifier Resolution Using Scope Chains

Identifier lookup follows a **scope chain**:

1. Search current (innermost) scope
2. Move outward through parent scopes
3. Stop at the global scope

The **first match** found is the valid binding.

---

4. Shadowing Rules to Implement

4.1 Allowed Rules

- Shadowing is allowed in **inner scopes**
- Each scope may contain unique identifiers

4.2 Disallowed Rules

- Redeclaration in the **same scope** is illegal

4.3 Shadowing Detection Categories

Category	Action
Safe Shadowing	Allowed silently
Warning-Worthy Shadowing	Allowed but compiler emits a warning
Illegal Shadowing	Compilation error

4.4 Specific Shadowing Cases

Shadowed Entity	Classification
Local shadows another local (outer block)	Safe
Local shadows function parameter	Warning
Local shadows global variable	Warning
Local shadows class field	Warning
Redeclaration in same scope	Illegal

---

5. Shadowing Detection Algorithm

5.1 Algorithm Overview

Shadowing detection is performed **during symbol insertion**.

5.2 Step-by-Step Algorithm

1. Let currentScope be the active scope
2. When inserting symbol s:
   - Check if s.name exists in currentScope
     - If yes → **Error: redeclaration**
   - Traverse parent scopes:
     - If same name is found:
       - Classify shadowing type
       - Emit warning if necessary
3. Insert symbol into currentScope

---

6. Pseudocode Implementation

6.1 Entering a New Scope

```
function enterScope(scopeType):
   newScope = createScope(scopeType)
```

```
   newScope.parent = currentScope
   currentScope = newScope
```

6.2 Exiting a Scope

```
function exitScope():
   currentScope = currentScope.parent
```

6.3 Inserting a Symbol

```
function insertSymbol(name, type, kind):
   if currentScope.contains(name):
      reportError("Redeclaration of identifier: " + name)
      return

   shadowedSymbol = lookupInOuterScopes(name)

   if shadowedSymbol != null:
      handleShadowing(name, shadowedSymbol, kind)

   currentScope.add(name, type, kind)
```

6.4 Detecting Shadowing

```
function lookupInOuterScopes(name):
   scope = currentScope.parent
   while scope != null:
      if scope.contains(name):
         return scope.get(name)
      scope = scope.parent
   return null
```

6.5 Emitting Warnings

```
function handleShadowing(name, shadowedSymbol, newKind):
   if shadowedSymbol.kind == PARAMETER:
      emitWarning("Variable '" + name +
               "' shadows function parameter")

   else if shadowedSymbol.kind == GLOBAL:
      emitWarning("Variable '" + name +
               "' shadows global variable")

   else if shadowedSymbol.kind == FIELD:
      emitWarning("Variable '" + name +
               "' shadows class field")

   else:
      // safe shadowing
      return
```

7. Warning Examples

7.1 Local Variable Shadows Global Variable

```
int count;

void func() {
    int count;   // shadows global variable
}
```

Compiler Warning:

Warning: Variable 'count' shadows global variable declared at line 1.

7.2 Block Variable Shadows Function Parameter

```
void sum(int x) {
    if (x > 0) {
        int x;   // shadows function parameter
    }
}
```

Compiler Warning:

Warning: Variable 'x' shadows function parameter.

---

8. Integration with Semantic Analysis

8.1 Type Checking

- Shadowed identifiers must still obey **type consistency**
- Type checking uses the **resolved symbol from the nearest scope**

8.2 Attribute Grammars

- Scope and symbol information are propagated as **inherited attributes**
- Type and binding information are **synthesized attributes**
- Shadowing detection occurs during attribute evaluation

8.3 AST Traversal

- On entering AST nodes (function, block): enterScope()
- On exiting nodes: exitScope()
- On variable declaration nodes: insertSymbol()

---

9. Conclusion

Implementing shadowing rules strengthens semantic analysis by:

- Preventing illegal redeclarations
- Identifying subtle logic errors early
- Improving program clarity and maintainability

By integrating shadowing detection into the symbol table mechanism, the compiler can issue meaningful warnings without violating language rules, thereby ensuring **robust and reliable semantic validation** before code generation.