Bahirdar University Faculty of Computing Software Engineering Copiler Design Assignment 3 2018 E.C

Abebaw Abebe --------BDU1504734

Submitted to: M.r Wondimu B.
Subition Dtae: 27-04-2018

Table of Contents Pages

Semantic Analysis: Scope & Symbol Table

Detecting Illegal Access to Private Members

1. Introduction

In compiler design, writing a program that is syntactically correct does not guarantee that the program is correct or safe to execute. Syntax analysis only verifies whether the program follows the grammatical rules of the programming language, such as correct placement of keywords, symbols, and structure. However, a program may still contain logical and meaning-related errors even if its syntax is valid. These deeper checks are performed during the semantic analysis phase of the compiler.

One of the most important responsibilities of semantic analysis is enforcing access control rules defined by the programming language. This includes detecting illegal access to private fields and methods in object-oriented programs. Proper enforcement of these rules requires a clear understanding of scope and an accurate symbol table. Without semantic analysis, programs could freely access private data, breaking encapsulation and leading to unsafe and unreliable software.

2. Scope in Semantic Analysis

2.1 What is Scope?

Scope defines the region of a program where an identifier, such as a variable, method, or class field, is visible and can be legally accessed. In object-oriented programming languages, scope exists at multiple levels. These include global scope, which applies to identifiers accessible throughout the program; class scope, which limits access to class members; method scope, which applies within a function or method; and block scope, which applies inside conditional statements and loops. Some languages also support nested class scope, where classes are defined inside other classes.

In addition to these structural scopes, access modifiers such as public, private, and protected further restrict how identifiers can be accessed. These modifiers define visibility rules that work together with scope to control access to program elements.

2.2 Why Scope Matters

Scope rules are essential because they allow the compiler to determine exactly which declaration an identifier refers to at any point in the program. This prevents ambiguity and name conflicts when multiple identifiers share the same name but exist in different scopes. Scope checking also enforces access restrictions, ensuring that private members are not accessed outside their allowed context. By enforcing scope rules, the compiler helps maintain program correctness, protects sensitive data, and improves overall program security.

3. Symbol Table in Semantic Analysis

3.1 What is a Symbol Table?

A symbol table is a central data structure used by the compiler to store detailed information about all identifiers declared in a program. Each entry in the symbol table represents one identifier and contains important attributes such as the identifier's name, data type, scope level, and access modifier. For object-oriented languages, symbol table entries also store information about class ownership, inheritance relationships, and nesting details.

The symbol table is continuously updated as the compiler processes declarations and enters or exits different scopes. This allows the compiler to keep track of which identifiers are currently visible and how they may be accessed.

3.2 Role of the Symbol Table in Access Control

The symbol table plays a critical role in enforcing access control rules during semantic analysis. It enables the compiler to determine whether an identifier is private, public, or protected and whether the current access context is valid. By consulting the symbol table, the compiler can decide if a particular field or method can be accessed from a given scope. Without a symbol table, it would be impossible for the compiler to detect illegal access to private members or enforce encapsulation rules.

4. Private Members and Access Rules

4.1 What are Private Members?

Private members are variables or methods of a class that are intended to be accessed only within the class where they are declared. They are not visible outside the class and cannot be accessed directly by external code. This restriction is a key aspect of encapsulation, one of the core principles of object-oriented programming. Encapsulation helps hide internal implementation details, allowing a class to control how its data is accessed and modified.

4.2 Access Rules for Private Members

Access to private members is strictly limited. A private member can always be accessed within its own class. In some programming languages, private members may also be accessed by nested classes or special friend-like structures. However, private members cannot be accessed from other unrelated classes, external functions, or subclasses in many languages. These rules ensure that private data remains protected and can only be manipulated in controlled ways.

5. Detecting Illegal Access Using Semantic Analysis

5.1 How the Compiler Performs the Check

During semantic analysis, the compiler checks every access to a class member. First, it identifies the member being accessed and looks it up in the symbol table. The compiler then examines the access modifier associated with the member and determines the current access context, such as the class or function where the access occurs. If the access context does not satisfy the language's access rules, the compiler reports a semantic error indicating illegal access to a private member. This checking is performed after syntax analysis and before code generation, ensuring that invalid programs do not proceed further.

During semantic analysis, the compiler:

1. Identifies the accessed member
2. Looks up the member in the **symbol table**
3. Checks its access modifier
4. Compares the access context (current class or scope)
5. Reports an error if access is not allowed

6. Practical Example: Illegal Access to a Private Field

Consider a program in which a class Student defines a private field age and provides a public method to access it safely. When the main() function attempts to directly modify age, the compiler flags an error. During semantic analysis, the symbol table marks age as a private member belonging to the Student class. Since main() exists outside this class, the semantic analyzer determines that the access violates scope and access control rules and reports an illegal access error. This prevents external code from directly modifying internal class data.

Example Code (Conceptual C++-like)

```
class Student {
private:
   int age;

public:
   int getAge() {
      return age;
   }
};

int main() {
   Student s;
   s.age = 20;
}
```

Semantic Analysis Explanation

- ✓ age is marked as private in the symbol table
- ✓ main() is outside the Student class
- ✓ Semantic analyzer detects that age is accessed from an invalid scope
- ✓ Compiler reports: **illegal access to private member**

7. Nested Classes and Private Member Access

7.1 Nested Class Example

Nested classes introduce more complex access scenarios. When a class is defined inside another class, the symbol table records this nesting relationship. If the nested class attempts to access a private member of the outer class, the semantic analyzer checks the language's rules to determine whether such access is permitted. In languages that allow nested classes to access private members of the enclosing class, the access is allowed. Otherwise, the compiler reports a semantic error. This demonstrates how semantic analysis must carefully consider both scope and language-specific rules.

```
class Outer {
private:
   int x;

public:
   class Inner {
   public:
      void show(Outer obj) {
         cout << obj.x;  // Allowed in some languages
      }
   };
};
```

Semantic Explanation

- ✓ x belongs to Outer
- ✓ Inner is nested inside Outer
- ✓ Symbol table records nesting relationship
- ✓ Semantic analyzer checks if nested access is allowed
- ✓ If allowed by the language, access is permitted
- ✓ Otherwise, a semantic error is reported

8. Friend-like Structures and Special Access

8.1 Concept of Friend Access

Some programming languages support friend-like mechanisms that grant special access privileges. A friend class or function is explicitly allowed to access the private members of another class. The symbol table records these friend relationships during compilation. When a private member is accessed, the semantic analyzer checks whether the accessing entity is listed as a friend. If so, access is permitted; if not, the compiler reports an illegal access error. This controlled exception allows flexibility while still preserving encapsulation.

Example

```
class Account {
private:
   int balance;

   friend class Bank;
};

class Bank {
public:
   void showBalance(Account a) {
      cout << a.balance;  // Allowed
   }
};
```

Semantic Analysis Handling

- Symbol table records Bank as a friend of Account
- When balance is accessed:
  - ✓ Semantic analyzer checks friend relationship
  - ✓ Access is allowed

Without the friend declaration, this would be an illegal access

9. How the Symbol Table Stores Access Information

The symbol table stores detailed access-related information for each class member, including its name, data type, scope, access level, and owning class. When a member is accessed, the compiler compares the owner class with the current access context. This comparison allows the compiler to accurately enforce access rules and detect violations early in the compilation process.

A simplified symbol table entry for a private member:

Name	Type	Scope	Access	Owner Class
age	int	Class	private	Student

During access:

- ✓ Compiler checks **Owner Class**
- ✓ Compares with **Current Access Context**
- ✓ Enforces access rules

10. Real-World Applications

Detecting illegal access to private members has significant real-world importance. In software security, it prevents unauthorized access to sensitive data and protects critical system logic. In large-scale software systems, it enforces modular design and reduces the risk of accidental misuse of internal class details. From a compiler reliability perspective, early detection of access violations prevents runtime crashes and logical failures. In object-oriented programming, these checks strongly support encapsulation and promote clean, maintainable, and robust code.

11. Role in the Overall Compiler Design Process

Illegal access detection occurs during the semantic analysis phase of the compiler pipeline. After lexical analysis performs tokenization and syntax analysis validates program structure, semantic analysis handles scope checking, symbol table lookup, and access control enforcement. Only programs that pass these semantic checks are allowed to proceed to intermediate code generation, optimization, and final code generation. This ensures that only semantically valid, secure, and meaningful programs are translated into executable code.

Conclusion

In semantic analysis, effective management of scope and symbol tables is essential for enforcing access control rules in object-oriented programs. By carefully tracking scope information and access modifiers, the compiler can accurately detect illegal access to private fields and methods, even in complex situations involving nested classes and friend-like structures. This process preserves encapsulation, enhances security, and ensures program correctness, making it a critical and indispensable part of the compiler design process.