Aufgabe A5.2 – Aufbau der Symboltabelle

November 21, 2025

Aufgabe A5.2 – Aufbau der Symboltabelle

Zielsetzung

In dieser Aufgabe geht es darum, den Aufbau einer Symboltabelle zu beschreiben, die für die semantische Analyse eines Programms erforderlich ist. Dies umfasst das Binden von Bezeichnern (z.B. Variablen und Funktionen) in den jeweiligen Gültigkeitsbereichen (Scopes) und das Auflösen von Bezeichnern beim Zugriff in verschachtelten Scopes.

1. Struktur der Symboltabelle

Die Symboltabelle verwaltet Informationen über Bezeichner (z.B. Variablen, Funktionen) und deren zugehörige Metadaten wie Typen, Adressen und Scopes. Für die Implementation einer Symboltabelle benötigen wir eine Datenstruktur, die für jedes Symbol (z.B. eine Variable oder eine Funktion) die folgenden Informationen speichert:

- **Name des Symbols**
- **Typ des Symbols**
- **Scope des Symbols** (welcher Scope dieses Symbol definiert hat)
- **Zusätzliche Informationen** (z.B. für Funktionen: Parameter, Rückgabewert)

Die Symboltabelle wird hierarchisch organisiert. Jeder Scope (z.B. ein Block, eine Funktion) erhält eine eigene Tabelle, die eine Verlinkung zum übergeordneten Scope enthält.

2. Hierarchische Organisation der Scopes

Scopes sind oft hierarchisch angeordnet:

- **Globaler Scope**: Der oberste Scope, in dem globale Variablen und Funktionen definiert sind.
- **Lokale Scopes**: Diese entstehen in Blöcken und Funktionen, z.B. durch die Eingabe von Codeblöcken { } oder Funktionsdeklarationen.
- **Verschachtelte Scopes**: Ein Block innerhalb einer Funktion oder ein Block innerhalb eines anderen Blocks erzeugt einen weiteren Scope.

Jeder Scope ist mit einer Symboltabelle verknüpft. Diese Symboltabelle enthält eine Liste von Symbolen, die im jeweiligen Scope definiert sind, sowie einen Verweis auf den übergeordneten Scope.

3. Methoden der Symboltabelle

Bind: Eine Funktion, die ein Symbol in die aktuelle Symboltabelle einfügt. Dabei wird sichergestellt, dass keine doppelten Bezeichner im gleichen Scope existieren (z.B. keine doppelte Variablendeklaration).

```
public class SymbolTable {
    private final Map<String, Symbol> symbols;
    private final SymbolTable enclosingScope;

    public SymbolTable(SymbolTable enclosingScope) {
        this.symbols = new HashMap<>();
        this.enclosingScope = enclosingScope;
    }

    public void bind(Symbol symbol) {
        if (symbols.containsKey(symbol.getName())) {
            throw new SemanticException("Symbol '" + symbol.getName() + "' is
        }
        symbols.put(symbol.getName(), symbol);
    }

    public Symbol resolve(String name) {
        Symbol symbol = symbols.get(name);
        if (symbol != null) {
            return symbol;
        }
        if (enclosingScope != null) {
            return enclosingScope.resolve(name);
        }
        return null; // Symbol nicht gefunden
    }
}
```

Resolve: Diese Methode sucht nach einem Symbol in der aktuellen Tabelle und, falls es dort nicht gefunden wird, in den übergeordneten Scopes.

4. Symboltypen

Wir definieren verschiedene Symboltypen, die in der Symboltabelle abgelegt werden. Es gibt zwei Hauptarten von Symbolen, die in diesem Fall relevant sind:

- **Variablen-Symbole** (für Variablen, die in einem Scope definiert werden)
- **Funktions-Symbole** (für Funktionen, die in einem Scope definiert werden)

Variablen-Symbol

```
public class VariableSymbol extends Symbol {
    private final Type type;

    public VariableSymbol(String name, Type type) {
        super(name);
        this.type = type;
    }

    public Type getType() {
        return type;
    }
}
```

Funktions-Symbol

```
public class FunctionSymbol extends Symbol {
    private final Type returnType;
    private final List<VariableSymbol> parameters;

    public FunctionSymbol(String name, Type returnType, List<VariableSymbol
        super(name);
        this.returnType = returnType;
        this.parameters = parameters;
    }

    public Type getReturnType() {
        return returnType;
    }

    public List<VariableSymbol> getParameters() {
        return parameters;
    }
}
```

5. Implementierung der Symboltabelle im Kontext eines Programms

Die Symboltabelle wird als Baumstruktur implementiert, wobei jeder Scope eine eigene Symboltabelle darstellt und eine Verlinkung zum übergeordneten Scope besteht. In einem typischen Compiler durchläuft der Visitor/Listener beim Verarbeiten des Parse-Trees die Regeln und ruft die entsprechenden Methoden auf, um Symbole zu binden.

Hier ein Beispiel, wie beim Verarbeiten von Variablen und Funktionen die Symbole in den jeweiligen Scopes gebunden werden:

```
public class SymbolTableBuilder extends MiniCBaseVisitor<Void> {
    private SymbolTable currentScope;
```

```
@Override
public Void visitProgram(MiniCParser.ProgramContext ctx) {
    currentScope = new SymbolTable(null); // Globaler Scope
    for (MiniCParser.StmtContext stmt : ctx.stmt()) {
        visit(stmt);
    }
    return null;
}

@Override
public Void visitVardecl(MiniCParser.VardeclContext ctx) {
    String name = ctx.ID().getText();
    Type type = resolveType(ctx.type().getText());
    VariableSymbol symbol = new VariableSymbol(name, type);
    currentScope.bind(symbol);
    return null;
}

@Override
public Void visitFndecl(MiniCParser.FndeclContext ctx) {
    String name = ctx.ID().getText();
    Type returnType = resolveType(ctx.type().getText());
    List<VariableSymbol> parameters = new ArrayList<>();
    if (ctx.params() != null) {
        for (MiniCParser.ParamContext paramCtx : ctx.params().param())
            String paramName = paramCtx.ID().getText();
            Type paramType = resolveType(paramCtx.type().getText());
            parameters.add(new VariableSymbol(paramName, paramType));
        }
    }
    FunctionSymbol functionSymbol = new FunctionSymbol(name, returnType
    currentScope.bind(functionSymbol);
    return null;
}

private Type resolveType(String typeName) {
    switch (typeName) {
        case "int":
            return new Type("int");
        case "string":
            return new Type("string");
        case "bool":
            return new Type("bool");
        default:
            throw new SemanticException("Unknown type " + typeName);
    }
}
```

}

6. Fehlerbehandlung

- **Doppelte Deklarationen**: Wenn ein Bezeichner bereits im aktuellen Scope vorhanden ist, wird eine `SemanticException` geworfen.
- **Nicht definierte Bezeichner**: Wenn ein Bezeichner nicht im aktuellen Scope und seinen Eltern gefunden werden kann, wird ebenfalls eine Ausnahme ausgelöst.