Bahir Dar University BiT

Faculty of Computing Department Of Software Engineering

Course: Principle Of Compiler Design

Assignment 62: Enforce Completeness of Switch Statements for Enums

By: Zefasil Mulu(ID: 1508437)

Submitted to: Lecturer Wondimu B

Submission date: 27/04/2018 E.C Bahir Dar, Ethiopia

Problem Description

In many programming languages, `enum` (enumeration) types define a **finite and fixed set of constant values**.
When a `switch` statement operates on an enum-typed expression, **semantic correctness requires that all possible enum values are handled**.

A switch over an enum is considered **complete** if **either**:

1. Every enum constant is explicitly covered by a `case` label, **or**
2. A `default` branch is present.

Failure to ensure completeness may result in **undefined behavior**, **unhandled execution paths**, or **logic errors**, even though the program may be syntactically valid.

Objective

The objective of this assignment is to **design and implement a semantic analysis pass** that:

- Identifies switch statements whose controlling expression is of an enum type
- Analyzes the enum definition to determine all possible enum values
- Verifies that:
  - All enum values are covered by `case` labels, **or**
  - A `default` branch exists
- Reports a **semantic error** if the switch statement is incomplete

This check is performed **during semantic analysis**, after type resolution and symbol table construction.

Compiler Phase Placement

This check belongs to the **Semantic Analysis phase**, because:

- It depends on **type information** (the expression must be an enum)
- It requires **symbol table access** to retrieve enum definitions
- It validates **program meaning**, not syntax

Required prior passes:

- Enum declarations must be registered in the symbol table
- Switch expression types must already be resolved

Semantic Rule Definition

Formal Semantic Rule

Let:

- `E` be an enum type
- `EnumValues(E)` be the set of constants declared in enum `E`
- `CaseLabels(S)` be the set of enum constants used in the switch statement `S`

Then a switch statement `S` over enum `E` is **semantically correct if**:

```
CaseLabels(S) = EnumValues(E)
OR
S contains a default branch
```

Otherwise, report a semantic error.

Algorithm Design

High-Level Steps

1. Traverse the Abstract Syntax Tree (AST)
2. For each `switch` statement:
    - Determine the type of the switch expression
3. If the type is **not an enum**, skip the check
4. If the type **is an enum**:
    - Retrieve the enum definition from the symbol table
    - Collect all enum constants
    - Collect all case labels used in the switch
    - Check for presence of a `default` case
5. If no `default` exists:
    - Verify all enum constants are covered
6. Report missing enum cases as a semantic error

Detailed Semantic Pass Logic

Data Structures Used

- **Symbol Table**
    - Stores enum definitions and their values
- **Set<String>**
    - For enum values
    - For case labels in switch

Pseudocode Implementation

```
for each SwitchStatement node S in AST:
    exprType = typeOf(S.expression)

    if exprType is EnumType:
        enumDef = symbolTable.lookupEnum(exprType.name)
        allEnumValues = enumDef.values

        coveredCases = empty set
        hasDefault = false

        for each case C in S.cases:
            if C is DefaultCase:
                hasDefault = true
            else:
                coveredCases.add(C.enumValue)

        if not hasDefault:
            for each value v in allEnumValues:
                if v not in coveredCases:
                    report error:
                        "Incomplete switch on enum " + exprType.name +
                        ". Missing case: " + v
```

Example Scenarios

Example 1: Correct and Complete Switch

```
enum Color { RED, GREEN, BLUE }

switch (color) {
    case RED:   ...
    case GREEN: ...
    case BLUE:  ...
}
```

✓ Semantically valid
All enum values are covered.

Example 2: Correct via Default Case

```
switch (color) {
    case RED: ...
    case GREEN: ...
    default: ...
}
```

✓ Semantically valid
`default` ensures completeness.

Example 3: Semantic Error (Incomplete)

```
switch (color) {
    case RED: ...
    case GREEN: ...
}
```

✗ Semantic Error

```
Error: Incomplete switch on enum Color. Missing case: BLUE
```

Error Reporting Strategy

Error messages should be:

- Clear and descriptive
- Reference the enum name
- Explicitly list missing enum values

Example Error Message

```
Semantic Error: Switch statement over enum 'Color' is incomplete.
Missing enum constant: BLUE
```

Edge Cases Considered

- Duplicate enum cases → handled by a different semantic rule
- Non-enum switch expressions → ignored by this pass
- Nested switches → handled naturally via AST traversal
- Empty enum → trivially complete

Why This Check Is Important

- Prevents silent logic errors
- Enforces exhaustive handling of enum values
- Improves code safety and maintainability
- Aligns with modern compiler practices (Java, Rust, Swift)

Conclusion

This semantic pass ensures that **switch statements over enum types are exhaustive**.
By analyzing enum definitions and switch-case coverage, the compiler can detect **missing execution paths at compile time**, significantly improving program correctness.

This solution integrates naturally into the semantic analysis phase and demonstrates strong understanding of **type systems, symbol tables, and control-flow semantics**, making it suitable for confident academic submission.