Bahir Dar University Bahir Dar Institute of Technology Faculty of Computing DEPARTMENT OF INFORMATION TECHNOLOGY

Principles of Compiler Design

Assignment 03: Semantic Rules & Error Detection

Student Name: YARED MESFIN
Student ID: 1411368
Course: Compiler Design
Assignment Topic: Enforcing Completeness of Switch Statements for Enums
Date: December 17, 2025

Introduction

The objective of this assignment is to implement a semantic analysis pass that ensures the correctness of switch statements involving enumerated types (Enums). In robust software development, it is critical that a switch statement covering an enum handles every possible value defined in that enum, or provides a default case. Failure to do so can lead to unhandled states and runtime errors.

This report documents the theoretical background algorithmic approach and C++ implementation used to detect and report "Incomplete Switch Statement" errors.

Problem Statement

Task: Create a semantic pass to check that switch statements over enum types cover all enum values or explicitly include a default branch.

Scenario:
Given an Enum definition (e.g., Traffic Light { RED, YELLOW, GREEN }) and a switch statement operating on a variable of this type:

- The compiler must verify that cases exist for RED, YELLOW, and GREEN.
- If a case is missing (e.g., GREEN is missing) and there is no default block the compiler must raise a semantic error.

Theoretical Background

Semantic Analysis is the third phase of the compiler design process. While Syntax Analysis ensures the code follows the grammatical rules (e.g., "are the parentheses balanced?") Semantic Analysis checks if the code conveys a valid meaning.

For this specific task, we utilize **Symbol Table** concepts to store the definition of the Enum (its allowed values). When the parser encounters a switch statement it acts as a consumer of this symbol table information to validate the completeness (exhaustiveness) of the logic.

Algorithm Design

The verification process follows these logical steps:

- ❖ **Input Parsing:** Identify the switch statement and the type of variable being evaluated.
- ❖ **Symbol Table Lookup:** Retrieve the full list of constants defined for that Enum type.
- ❖ **Case Collection:** Iterate through the switch body and collect all case labels into a set.

- **Default Check:**
    - If a default label exists the switch is marked **Valid** (Safe).
    - If no default exists proceed to Step 5.
- **Completeness Check:**
    - Calculate the set difference:
        - Missing=AllEnumValues CoveredCasesMissing=AllEnumValues CoveredCases.
    - If the Missing set is empty the switch is **Valid**.
    - If the Missing set is not empty report an **Error** listing the missing values.

Implementation (C++)

The following C++ program demonstrates the semantic pass. It simulates the AST (Abstract Syntax Tree) structures for Enums and Switch statements and implements the validation logic described above.

```
#include <iostream>
#include <vector>
#include <string>
#include <set>
#include <algorithm>

using namespace std;

// --- AST Structures ---

// Represents the definition of an Enum in the Symbol Table
struct EnumDefinition {
    string name;
    set<string> distinctValues; // Example: {"RED", "YELLOW", "GREEN"}
};

// Represents a Switch Statement node in the AST
struct SwitchStatement {
    string variableType;        // The Enum type being switched on
    vector<string> caseLabels;  // Cases implemented by the user
    bool hasDefault;            // True if 'default:' block exists
};

// --- Semantic Analyzer Module ---
```

```
class SemanticAnalyzer {
public:
    /**
     * Checks if a switch statement is exhaustive (covers all enum values).
     * @param enumDef The definition of the enum from the Symbol Table.
     * @param switchStmt The switch statement to validate.
     */
    void checkSwitchCompleteness(const EnumDefinition& enumDef, const
SwitchStatement& switchStmt) {

        cout << "Analyzing switch on Enum '" << enumDef.name << "'..." << endl;

        // Rule 1: Existence of Default
        // If a default case exists, it handles all unspecified values.
        if (switchStmt.hasDefault) {
            cout << "  [STATUS]: Valid. (Switch contains a default case)." << endl << "-----------
---------------------------" << endl;
            return;
        }

        // Rule 2: Exhaustive Coverage
        // If no default, we must ensure every enum value has a corresponding case.
        set<string> casesCovered(switchStmt.caseLabels.begin(),
switchStmt.caseLabels.end());
        vector<string> missingValues;

        for (const string& val : enumDef.distinctValues) {
            // Check if the enum value is missing from the covered cases
            if (casesCovered.find(val) == casesCovered.end()) {
                missingValues.push_back(val);
            }
        }

        if (missingValues.empty()) {
            cout << "  [STATUS]: Valid. (All enum values are explicitly covered)." << endl;
        } else {
            // Report Semantic Error
            cout << "  [ERROR]: Incomplete switch statement!" << endl;
            cout << "  [DETAILS]: The following cases are missing: ";
            for (const string& missing : missingValues) {
                cout << "[" << missing << "] ";
            }
            cout << endl;
        }
        cout << "-------------------------------------" << endl;
    }
};

// --- Main Execution (Test Driver) ---

int main() {
    SemanticAnalyzer analyzer;

    // define the Enum in our "Symbol Table"
    EnumDefinition trafficLight;
    trafficLight.name = "TrafficLight";
    trafficLight.distinctValues = {"RED", "YELLOW", "GREEN"};

    // TEST CASE 1: Incomplete Switch (Error Expected)
    // User forgot 'GREEN' and didn't provide a default.
    SwitchStatement badSwitch;
    badSwitch.variableType = "TrafficLight";
    badSwitch.caseLabels = {"RED", "YELLOW"};
    badSwitch.hasDefault = false;
    analyzer.checkSwitchCompleteness(trafficLight, badSwitch);

    // TEST CASE 2: Complete Switch (Valid)
    // User included all 3 values.
    SwitchStatement goodSwitch;
    goodSwitch.variableType = "TrafficLight";
    goodSwitch.caseLabels = {"GREEN", "RED", "YELLOW"};
    goodSwitch.hasDefault = false;
    analyzer.checkSwitchCompleteness(trafficLight, goodSwitch);

    // TEST CASE 3: Partial Switch with Default (Valid)
    // User only handled RED, but provided a default for the rest.
    SwitchStatement defaultSwitch;
    defaultSwitch.variableType = "TrafficLight";
    defaultSwitch.caseLabels = {"RED"};
    defaultSwitch.hasDefault = true;
    analyzer.checkSwitchCompleteness(trafficLight, defaultSwitch);

    return 0;
}
```

Test Results & Output Analysis

The program was executed with three distinct test cases to verify robust error detection.

Output:

```
codeText
Analyzing switch on Enum 'TrafficLight'...
  [ERROR]: Incomplete switch statement!
  [DETAILS]: The following cases are missing: [GREEN]
--------------------------------------
Analyzing switch on Enum 'TrafficLight'...
  [STATUS]: Valid. (All enum values are explicitly covered).
--------------------------------------
Analyzing switch on Enum 'TrafficLight'...
  [STATUS]: Valid. (Switch contains a default case).
--------------------------------------
```

Discussion:

1. **Test Case 1** correctly identified that GREEN was missing and flagged an error. This confirms the logic detects incompleteness.
2. **Test Case 2** passed because the number of cases matched the number of enum definitions exactly.
3. **Test Case 3** passed because the default flag short-circuited the check, adhering to standard C++ semantic rules.

Conclusion

In this assignment, I successfully implemented a Semantic Analysis module that enforces completeness for switch statements over Enums. This ensures type safety and reduces potential runtime bugs caused by unhandled states. The implementation demonstrates the practical application of Symbol Tables and set operations within the compiler construction pipeline.