Name: Zelalem Getahun **ID: 1508462**

Bahir Dar University

Principles of Compiler Design on Syntax Analysis

Individual Assignment-3

Add boolean type checking to an expression language

Boolean type checking ensures that **logical operators** in a programming language (AND, OR, NOT) operate on **boolean operands only**. Type checking prevents semantic errors, such as applying arithmetic operators to boolean values or using logical operators on integers.

1. Type System Extension

Boolean type checking is part of **semantic analysis** in the compiler. It ensures that boolean expressions are used correctly and that logical operators such as `AND`, `OR`, and `NOT` are only applied to valid boolean operands. This prevents semantic errors that might otherwise cause incorrect program execution at runtime.

Initially, assume the expression language already supports numerical data types such as:

- **int**
- **float**

To enhance the language, we extend the type system to introduce:

- **bool**

This allows the language to handle logical decision-making constructs effectively.

Supported Boolean Operators and Rules

The following boolean operators are supported:

- **AND (&&)** → both operands must be boolean
- **OR (||)** → both operands must be boolean
- **NOT (!)** → operand must be boolean

If any of these rules are violated, the compiler must generate a **type error** during semantic analysis.

Formal Type Rules

Expression	Rule	Result Type
e1 && e2	type(e1) == bool AND type(e2) == bool	bool
e1 || e2	type(e1) == bool AND type(e2) == bool	bool
!e	type(e) == bool	bool
e1 + e2	type(e1), type(e2) ∈ {int, float}	int / float
e1 < e2	type(e1), type(e2) ∈ {int, float}	bool

These rules guarantee:

- Logical operators work only with **boolean values**
- Arithmetic operators work only with **numeric values**
- Relational operators return a **boolean** result

Purpose and Importance

Adding boolean type checking ensures:

- Prevention of invalid or meaningless logical operations
- Early detection of semantic errors before code generation
- More reliable and predictable programs
- Support for control statements like `if`, `while`, and `for`

Thus, boolean type checking strengthens the semantic correctness of the language and contributes to a robust compiler design.

2. Implementation (C++ / Pseudocode)

Here's an example of **type checking for boolean expressions** in a small expression language.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;
```

```
// Example type enum
enum Type { INT, FLOAT, BOOL, ERROR };

// Symbol Table (variable name -> type)
map<string, Type> symTable = {
    {"a", BOOL}, {"b", BOOL}, {"x", INT}, {"y", FLOAT}
};

// Function to check binary operations
Type checkBinary(Type left, Type right, string op) {
    if(op == "&&" || op == "||") {
        if(left == BOOL && right == BOOL) return BOOL;
        else {
            cout << "Type Error: Operator " << op
                 << " requires boolean operands.\n";
            return ERROR;
        }
    }
    if(op == "+" || op == "-" || op == "*" || op == "/") {
        if((left == INT || left == FLOAT) && (right == INT || right ==
FLOAT))
            return (left == FLOAT || right == FLOAT) ? FLOAT : INT;
        else {
            cout << "Type Error: Arithmetic operator " << op
                 << " requires numeric operands.\n";
            return ERROR;
        }
    }
    return ERROR;
}

// Function to check unary operations
Type checkUnary(Type operand, string op) {
    if(op == "!") {
        if(operand == BOOL) return BOOL;
        else {
            cout << "Type Error: NOT operator requires boolean
operand.\n";
            return ERROR;
        }
    }
    return ERROR;
}

// Example usage
int main() {
    Type t1 = symTable["a"];
    Type t2 = symTable["b"];
    Type result = checkBinary(t1, t2, "&&");  // Correct usage
    cout << "Type of a && b = " << (result == BOOL ? "BOOL" : "ERROR")
<< "\n";

    Type t3 = symTable["x"];
    result = checkBinary(t1, t3, "&&");       // Error usage
}
```

3. Example Cases

To demonstrate boolean type checking in action, consider the following sample expressions evaluated using the symbol table:

```
a → bool
b → bool
x → int
y → float
```

The results of checking different boolean and mixed expressions are shown below:

Expression	Type Check Result	Explanation
a && b	**BOOL (Valid)**	Both operands are boolean, so logical AND is valid
a && x	**ERROR**	`x` is an integer, logical AND requires two boolean operands
!b	**BOOL (Valid)**	`b` is boolean, so NOT operation is legal
!y	**ERROR**	`y` is float, NOT can only be applied to boolean values

---

Detailed Interpretation

1. **a && b**
   - `a` is `bool`
   - `b` is `bool`
   - Since both operands are boolean, the result type is `bool`
   - ✓ Valid logical operation
2. **a && x**
   - `a` is `bool`
   - `x` is `int`
   - Logical AND cannot operate on integers
   - ✗ Semantic Type Error
3. **!b**
   - `b` is `bool`
   - NOT operator expects a boolean operand
   - ✓ Valid boolean expression
   - Result type is `bool`
4. **!y**
   - `y` is `float`
   - NOT cannot be applied to numeric values
   - ✗ Semantic Type Error

Why These Examples Matter

These examples prove that:

- Boolean operators enforce strict type discipline
- Invalid type combinations are detected before execution
- The compiler prevents logical misuse of different data types
- The type checker ensures program correctness and safety

4. Key Points

- Boolean type checking prevents logical errors in programs.
- Extending the type system allows compilers to **catch errors at compile-time**, avoiding runtime failures.
- The **symbol table** is essential for storing variable types.
- This approach can be extended to **functions, generics, and object-oriented constructs** for robust semantic checking.

References

1. Aho, A., Lam, M., Sethi, R., Ullman, J. "Compilers: Principles, Techniques, and Tools"
2. Behrouz Forouzan, "Compiler Construction Principles and Practice"
3. GeeksforGeeks – "Type Checking in Compilers"