Basic C Compiler Project Report

Maxwell Aboagye

25.01.2025

Contents

1 Language Description

This project implements a subset of the C programming language, focusing on fundamental operations and type checking. The compiler supports basic data types such as int and bool, arithmetic operations, control structures, and includes comprehensive error reporting with line numbers.

The key focus areas of the implementation are:

- Static type checking with clear error messages
- Symbol table management using dynamic data structures
- Support for both positive and negative numbers
- Control flow structures (if-else, while)
- Basic arithmetic and logical operations

2 Grammar Implementation

The grammar implementation defines the language structure through a set of production rules. Here is a detailed breakdown of the grammar:

2.1 Program Structure

```
program -> statement_list
         |

statement_list -> statement
                | statement_list statement
```

2.2 Statements

```
statement -> declaration ;
           | assignment ;
           | if_statement
           | while_statement
           | print_statement ;
           | return_statement ;
```

2.3 Declarations and Types

The grammar supports two fundamental types:

```
type_specifier -> int | bool

declaration -> type_specifier IDENTIFIER
             | type_specifier IDENTIFIER = expression

assignment -> IDENTIFIER = expression
```

2.4 Expressions

Expression handling is hierarchical to ensure proper operator precedence:

```
expression -> simple_expression
            | expression AND simple_expression
            | expression OR simple_expression

simple_expression -> term
                   | simple_expression + term
                   | simple_expression - term
                   | simple_expression > term
                   | simple_expression < term
                   | simple_expression >= term
                   | simple_expression <= term
                   | simple_expression == term
                   | simple_expression != term

term -> factor
      | term * factor
      | term / factor

factor -> INT_LITERAL
        | BOOL_LITERAL
        | IDENTIFIER
        | ( expression )
        | NOT factor
        | - factor
```

3 Symbol Table Implementation

The symbol table is implemented as a dynamic data structure that maintains variable information throughout the execution of the program. The implementation is found in symbol_table.c and includes several key components:

3.1 Data Structures

```
typedef enum {
    TYPE_INT,
    TYPE_BOOL,
    TYPE_ERROR
} DataType;

typedef struct Symbol {
    char* name;
    DataType type;
    union {
        int int_val;
        bool bool_val;
    } value;
```

```
    struct Symbol* next;
} Symbol;

typedef struct {
    Symbol* head;
    Symbol* tail;
} SymbolTable;
```

3.2 Key Operations

3.2.1 Symbol Creation

```
Symbol* create_symbol(char* name, DataType type) {
    Symbol* symbol = (Symbol*)malloc(sizeof(Symbol));
    symbol->name = strdup(name);
    symbol->type = type;
    symbol->next = NULL;

    // Initialize default values
    if (type == TYPE_INT) {
        symbol->value.int_val = 0;
    } else if (type == TYPE_BOOL) {
        symbol->value.bool_val = false;
    }

    return symbol;
}
```

3.2.2 Symbol Lookup

```
Symbol* lookup_symbol(char* name) {
    Symbol* current = table->head;
    while (current != NULL) {
        if (strcmp(current->name, name) == 0) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}
```

4 Operations Implementation

The operations module (operations.c) handles all computations with comprehensive type checking:

4.1 Arithmetic Operations

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/) with zero division checking

Example implementation of addition:

```
Symbol *addition(Symbol *a, Symbol *b) {
    if (a->type != TYPE_INT || b->type != TYPE_INT) {
        fprintf(stderr, "Error at line %d: Cannot perform addition
           between '%s' (%s) and '%s' (%s)\n",
                line_number,
                a->name, get_type_name(a->type),
                b->name, get_type_name(b->type));
        exit(1);
    }

    Symbol *result = create_temp_result(TYPE_INT);
    result->value.int_val = a->value.int_val + b->value.int_val;
    return result;
}
```

4.2 Logical Operations

- AND (&&)
- OR (||)
- NOT (!)

4.3 Comparison Operations

All comparison operations include type compatibility checking:

- Greater than (>)
- Less than (<)
- Greater than or equal to (>=)
- Less than or equal to (<=)
- Equal to (==)
- Not equal to (!=)

5 Error Handling

The compiler implements comprehensive error handling with line number tracking:

5.1 Type Errors

- Mismatched types in assignments
- Invalid operation types
- Type conversion errors

5.2 Runtime Errors

- Division by zero
- Undefined variable usage
- Variable redeclaration

6 Examples

6.1 Valid Program

```
int x = 5;
int y = 10;
bool flag = true;

if (x < y) {
    print x;    // Will print 5
} else {
    print y;
}

while (flag) {
    x = x + 1;
    if (x >= 10) {
        flag = false;
    }
}

print x;    // Will print 10
```

6.2 Invalid Programs

6.2.1 Type Mismatch

```
int x = 5;
bool y = x;  // Error: Cannot initialize bool with int
```

6.2.2 Invalid Operation

```
bool flag = true;
bool other = false;
bool result = flag + other;  // Error: Cannot add booleans
```

7 Building and Running

7.1 Prerequisites

- GCC compiler
- Flex lexical analyzer
- Bison parser generator
- Make (optional)

7.2 Build Process

```
make clean
make

# Or manually:
flex lexer.l
bison -d parser.y
gcc lex.yy.c parser.tab.c symbol_table.c operations.c -o compiler
```

7.3 Execution

```
# Interactive mode
./compiler

# With input file
./compiler < input_file.txt
```

8 Limitations and Future Work

- Single scope implementation
- Limited to int and bool types
- No function support
- No arrays or pointers

9 Conclusion

This compiler project successfully implements a focused subset of C with efficient type check and error reporting. The implementation demonstrates core compiler concepts that include lexical analysis, parsing, and semantic analysis while maintaining a clean and modular codebase.

The project particularly excels in the following.

- Strong type checking
- Clear error reporting with line numbers
- Comprehensive operation handling
- Efficient symbol table management

Although this project has some limitations, it provides a solid foundation for future extensions and enhancements.