A Report on
Compiler Design Lab (CS304): Mini Project Phase 1

by

Shanjiv A (Roll No: 231CS155)

Prabhav S Korwar (Roll No: 231CS141)

Akshaj PVY (Roll No: 231CS109)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA
SURATHKAL, MANGALURU-575025
13-August-2025

1 Introduction

1.1 Lexical Analysis

Lexical analysis is the first phase of a compiler, which takes the source code as input and produces a sequence of tokens as output. A lexical analyzer, or scanner, reads the input characters of the source code, identifies the lexemes, and produces the corresponding tokens. This process is typically implemented using deterministic finite automata (DFA).

In this project, we have implemented a lexical analyzer for the C programming language using Flex, a scanner generator tool that converts a lexical specification into a C program that performs pattern matching on text. Our scanner recognizes standard C tokens, maintains symbol and constant tables, and provides detailed information about the code structure.

1.2 Tokens & Lexemes

A token is a categorized block of text in the source code. Each token has:

- A token type or category (e.g., keyword, identifier, operator)
- The actual string value or lexeme (e.g., "if", "main", "+")

Our scanner identifies the following token types:

- Keywords (if, while, return, etc.)
- Identifiers (variable and function names)
- Constants (numeric literals, string literals, character literals)
- Operators (+, -, *, /, etc.)
- Punctuation (;, , , etc.)
- Preprocessor directives (#include, #define)

2 Overview of the Code

Our lexical analyzer for the C programming language is implemented using Flex and provides comprehensive token recognition and tracking capabilities. The scanner is organized into several key components:

2.1 Key Components

- **Symbol Table**: An array-based implementation with efficient lookup that tracks identifiers, their types, dimensions, frequencies, and function-related information.
- **Constant Table**: An array-based implementation that tracks constant values with their variable names, line numbers, and types.
- **State Tracking**: Maintains context for declarations, assignments, and function argument collection using various state variables.

- **Lexical Rules**: Defines regular expressions for token patterns and associates actions with each pattern to maintain tables and context.
- **Error Handling**: Detects and reports invalid tokens with line numbers, allowing the scanner to find multiple errors in a single pass.

2.2 Data Structures

The main data structures used in the scanner are:

2.2.1 Symbol Structure

```
typedef struct Symbol {
    char* name;
    char* type;
    char* dimensions;
    int   frequency;
    char* return_type;
    char* param_lists;
    int   is_function;
} Symbol;
```

2.2.2 Constant Structure

```
typedef struct Constant {
    char* var_name;
    int   line;
    char* value;
    char* type;
} Constant;
```

3 Scanner Implementation

3.1 Key Algorithms

3.1.1 Symbol Table Implementation

We use a simple array-based implementation for the symbol table, providing efficient storage and lookup:

```
// Symbol table array implementation
Symbol symbol_table[MAX_SYMBOLS];
int symbol_count = 0;

// Simple lookup function
Symbol* lookup(const char* name) {
    for (int i = 0; i < symbol_count; i++) {
        if (strcmp(symbol_table[i].name, name) == 0) {
            return &symbol_table[i];
```

```
        }
    }
    return NULL;
}
```

3.1.2 Function Argument Collection

The scanner uses a state machine to collect function arguments:

1. When a function call is detected, enters `FUNCARGS` state
2. Collects all text between parentheses
3. Handles nested parentheses by tracking depth
4. When the closing parenthesis is found, processes the collected arguments

3.1.3 Variable Assignment Tracking

For tracking variable assignments:

1. When an identifier is found, it's stored in `last_ident`
2. When `=` is encountered after an identifier, `in_assignment` is set
3. The current variable name is stored in `current_var`
4. When a constant is found during assignment, it's associated with the variable

4 List of Recognized Tokens and Their Meaning

Token Type	**Examples**	**Description**
KEYWORD	`if`, `while`, `return`	C language reserved words
TYPE	`int`, `float`, `char`	Built-in C data types
IDENT	Variable and function names	Matches `[a-zA-Z_][a-zA-Z0-9_]*`
NUMBER	`123`, `3.14`	Integer and floating-point literals
STRING	`"Hello World"`	String literals in double quotes
CHAR	`'a'`	Character literals in single quotes
OP	`+`, `-`, `*`, `/`, `=`	Arithmetic and assignment operators
PUNCT	`;`, `{`, `}`, `(`, `)`	Punctuation marks and delimiters
PREPROC	`#include`, `#define`	Preprocessor directives
STRING	`"Hello World"`	Text between double quotes
CHAR	`'a'`, `'\n'`	Single characters with escape sequences
NUMBER	`123`, `0xFF`, `3.14`	Numeric literals
OP	`+`, `-`, `==`, `!=`	Arithmetic, comparison, logical operators
PUNCT	`;`, `{`, `}`, `(`	Separators and delimiters
PREPROC	`#include`, `#define`	C preprocessor directives

5 Deterministic Finite Automata (DFA) for Token Recognition

Our scanner is implemented using a collection of Deterministic Finite Automata (DFAs), each responsible for recognizing specific token patterns. This section presents the formal state diagrams for the key token recognition mechanisms.

5.1 Main Scanner States

The lexical analyzer operates in multiple states, with transitions triggered by specific character sequences. The figure below shows the primary scanner states and the transitions between them:

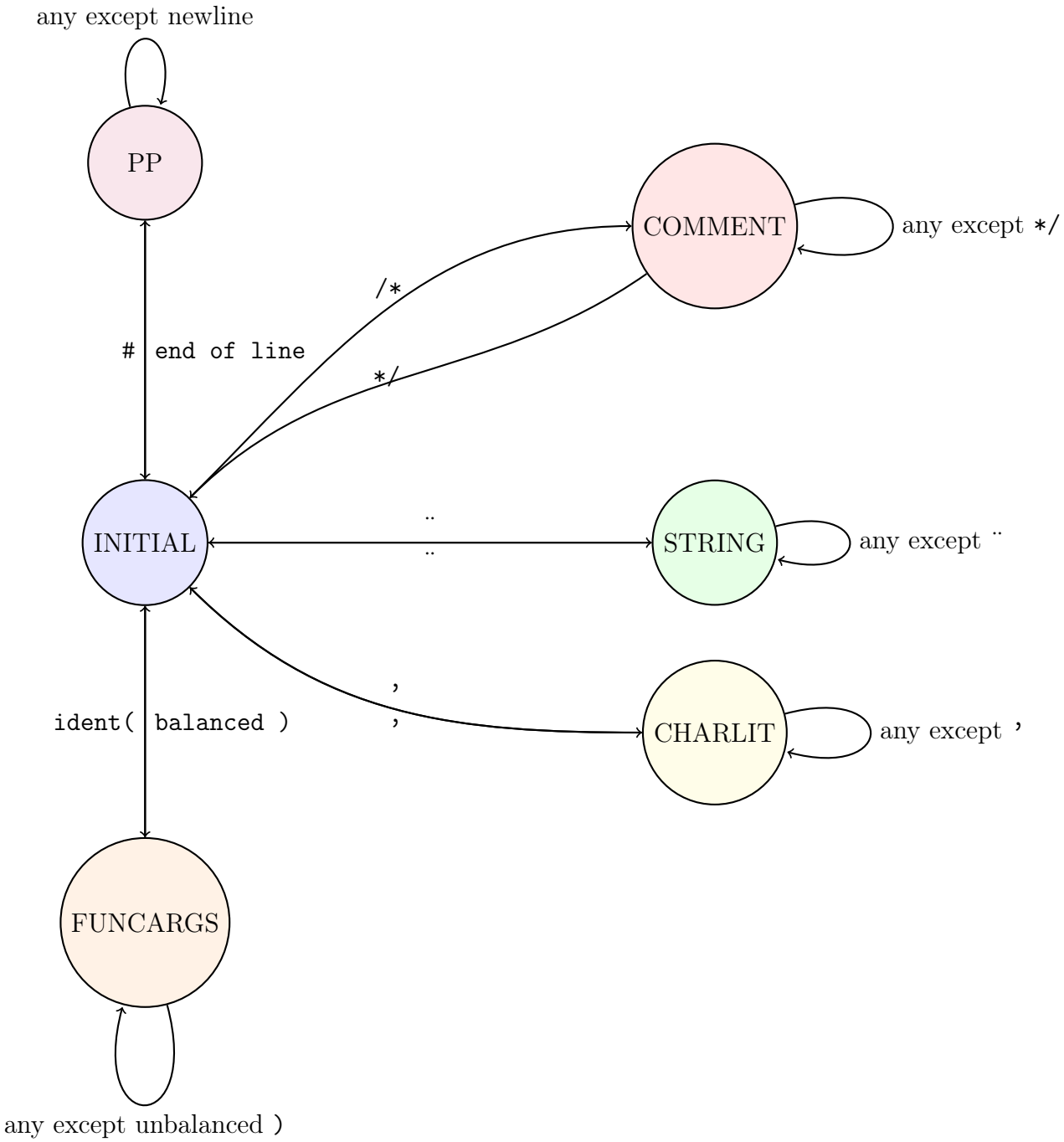


Figure 1: Main Flex states and transitions in the scanner

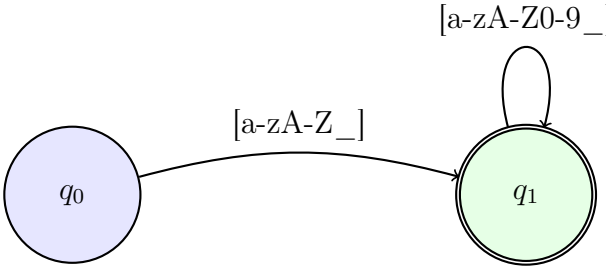
Each state handles a specific aspect of lexical analysis:

- **INITIAL**: Default state where most tokens are recognized
- **COMMENT**: Activated when entering comments, ignores all content until comment end
- **STRING**: Captures string literals between double quotes
- **CHARLIT**: Captures character literals between single quotes

- **PP**: Handles preprocessor directives starting with #
- **FUNCARGS**: Collects function arguments between parentheses

5.2 Identifier Recognition DFA

The DFA below formally describes the pattern used to recognize C identifiers. Identifiers must start with a letter or underscore, followed by zero or more letters, digits, or underscores.

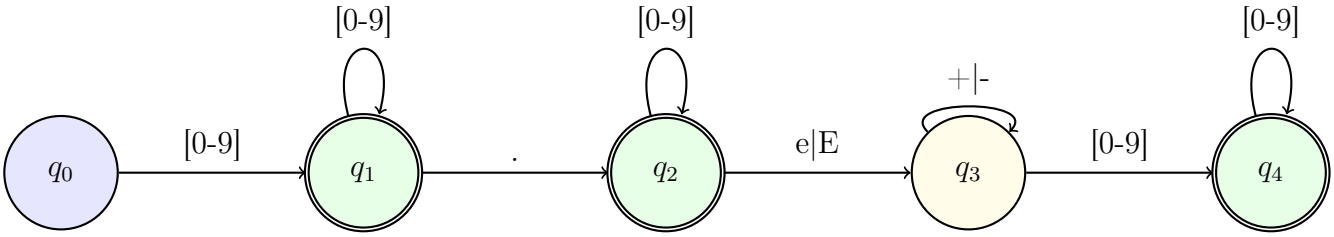


Pattern: [a-zA-Z_][a-zA-Z0-9_]*
Examples: main, x1, _count
Token: IDENT

Figure 2: DFA for identifier recognition with examples

5.3 Number Recognition DFA

The following DFA recognizes integers, floating-point numbers, and numbers in scientific notation. It includes states for the integer part, decimal part, and exponent part.



States:
q_0: Start
q_1: Integer part (accepting)
q_2: Fractional part (accepting)
q_3: Exponent sign
q_4: Exponent digits (accepting)

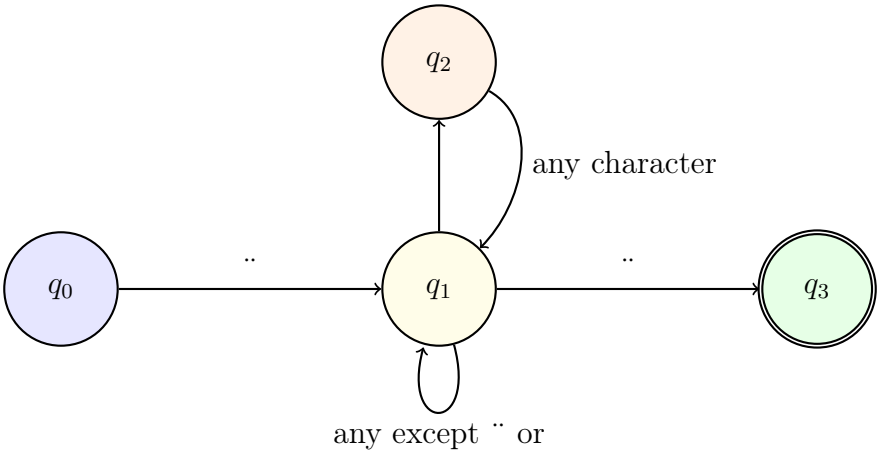
Figure 3: DFA for numeric literals recognition

Examples recognized by this DFA:

- **Integer literals:** 42, 123, 0
- **Floating-point literals:** 3.14, 0.5, 1.0
- **Scientific notation:** 1.2e3, 5.4E-2, 1e+10

5.4 String Literal Recognition DFA

The DFA below models the recognition of string literals. It handles escape sequences and ensures proper string termination.



States:
q_0: Start
q_1: Inside string
q_2: Escape sequence
q_3: End of string (accepting)

Figure 4: DFA for string literal recognition

This DFA recognizes:

- Simple strings: `"Hello world"`
- Strings with escape sequences: `"Line1\nLine2"`
- Empty strings: `""`

5.5 Comment Recognition DFA

The scanner recognizes both single-line and multi-line comments using the following DFA:

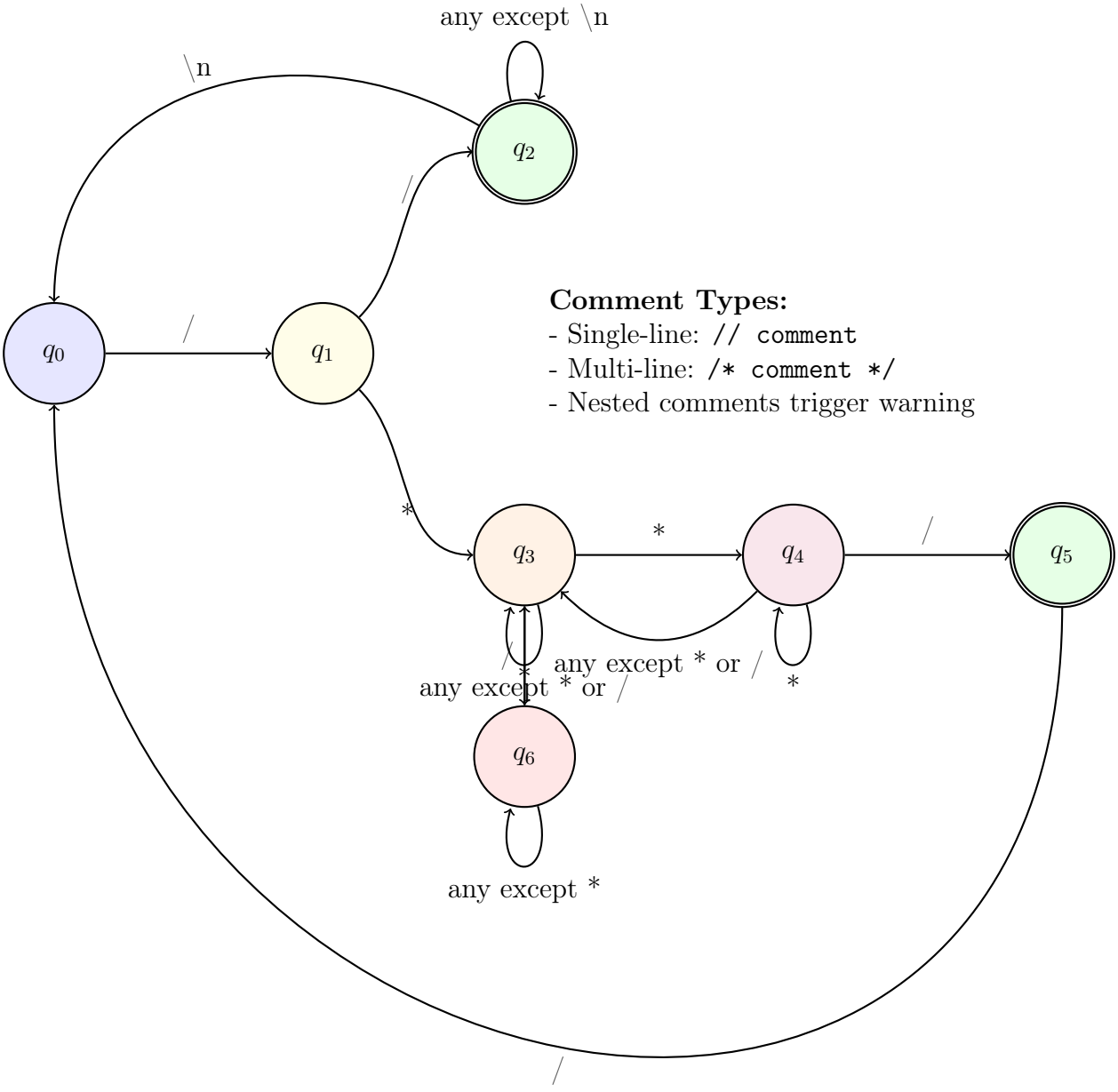


Figure 5: DFA for comment recognition

States in the comment DFA:

- q_0: Initial state
- q_1: Encountered first /

- q_2: In single-line comment
- q_3: In multi-line comment
- q_4: Potentially ending multi-line comment
- q_5: End of multi-line comment
- q_6: Potential nested comment start (triggers warning)

6 Assumptions Made Beyond the Basic Language Description

In implementing our scanner, we made several assumptions and design decisions that go beyond the basic C language specification:

1. **Multi-character Constants**: The scanner accepts multi-character constants like `'xy'` even though they're not standard C, treating them as valid character literals.
2. **Nested Comments**: The scanner detects attempted nested comments like `/* outer /* inner */ outer */` and issues a warning as they are not supported in standard C. Following the behavior of standard C compilers, it treats the first closing `*/` it encounters as the end of the comment.
3. **Identifier Length**: No limit is enforced on identifier length, allowing arbitrarily long identifiers.
4. **Type Qualifiers**: Type qualifiers like `const` and `volatile` are treated as regular keywords rather than as part of type declarations.
5. **Symbol Table Information**: The scanner tracks additional information beyond basic token recognition, including variable types, function parameters, array dimensions, and usage frequency.
6. **Variable-Constant Association**: The scanner associates constants with the variables they are assigned to, which is not a requirement of a basic lexical analyzer.

6.1 Test Case 1: Valid C Code

```
// Test Case 1: Valid C code
#include <stdio.h>
int main() {
    int a = 10;
    float b = 20.5;
    char c = 'x';
    printf("Hello World\n");
    return 0;
}
```

Output:

```
[line 2] PREPROC      : #include <stdio.h>
[line 3] TYPE         : int
[line 3] IDENT        : main
[line 3] PUNCT        : (
[line 3] PUNCT        : {
[line 4] TYPE         : int
[line 4] IDENT        : a
[line 4] OP           : =
[line 4] NUMBER       : 10
[line 4] PUNCT        : ;
...

==== SYMBOL TABLE ====
Name                 Type            Dimensions   Frequency  Return Type
a                    int             -            1          -
b                    float           -            1          -
c                    char            -            1          -
printf               -               -            1          -
main                 int             -            1          int

==== CONSTANT TABLE ====
Variable Name        Line No.   Value                          Type
a                    4          10                             int
b                    5          20.5                           float
c                    6          'x'                            char
-                    8          0                              int
```

6.2 Test Case 2: Invalid Tokens

```
// Test Case 2: Invalid tokens and errors
#include <stdio.h>
int main() {
    int a = 10$;
    float b = 20.5.3;
    char c = 'xy';
    printf("Hello World\n");
    return 0;
}
```

Output (showing error detection):

```
[line 4] TYPE         : int
[line 4] IDENT        : a
[line 4] OP           : =
[line 4] NUMBER       : 10
[line 4] ERROR: Invalid token '$'
[line 5] TYPE         : float
[line 5] IDENT        : b
```

```
[line 5] OP           : =
[line 5] ERROR: Invalid float literal '20.5.3'
```

6.3 Test Case 3: Comments

```
// Test Case 3: Comments and nested comments
#include <stdio.h>
/* This is a comment
   /* Nested comment */
   End of comment */
int main() {
    // Single line comment
    int x = 42;
    return 0;
}
```

Output (comments correctly ignored):

```
[line 2] PREPROC      : #include <stdio.h>
[line 6] TYPE         : int
[line 6] IDENT        : main
...
```

6.4 Test Case 4: Function Definitions and Calls

```
// Example function definition and calls
int multiply(int x, int y) {
    return x * y;
}

int main() {
    int product = multiply(6, 7);
    return 0;
}
```

Output (showing function tracking):

```
==== SYMBOL TABLE ====
Name                Type         Frequency  Return Type  Parameters Lists
multiply            int          2          int          int x, int y ; 6, 7
product             int          1          -            -
main                int          1          int          -
```

7 Test Cases

This section contains all the test cases used to validate our lexical analyzer.

7.1 Test Case 1: Valid C Code

This test case validates the scanner's ability to recognize basic C tokens in a valid program.

```
// Test Case 1: Valid C code
#include <stdio.h>
int main() {
    int a = 10;
    float b = 20.5;
    char c = 'x';
    printf("Hello World\n");
    return 0;
}
```

7.2 Test Case 2: Invalid Tokens and Errors

This test case checks the scanner's error detection for invalid tokens and malformed literals.

```
// Test Case 2: Invalid tokens and errors
#include <stdio.h>
int main() {
    int a = 10$;
    float b = 20.5.3;
    char c = 'xy';
    printf("Hello World\n");
    return 0;
}
```

7.3 Test Case 3: Comments and Nested Comments

This test case verifies how the scanner handles comments, including nested comments.

```
// Test Case 3: Comments and nested comments
#include <stdio.h>
/* This is a comment
   /* Nested comment */
   End of comment */
int main() {
    // Single line comment
    int x = 42;
    return 0;
}
```

7.4 Test Case 4: Preprocessor Directives and Function Call

This test case checks preprocessor directive recognition and function call tracking.

```
// Test Case 4: Preprocessor directives and function call
#define MAX 100
void foo(int a, float b) {
    int arr[MAX];
}
int main() {
    foo(10, 20.5);
    return 0;
}
```

7.5 Test Case 5: String and Character Literals with Escape Sequences

This test case validates the scanner's handling of character literals with escape sequences.

```
// Test Case 5: String and character literals
#include <stdio.h>
int main() {
    char c = '\n';
    char* str = "Test string with escape \" and newline\n";
    return 0;
}
```

7.6 Test Case 6: Advanced String and Character Literals

This test case validates more complex string and character literals, including various escape sequences.

```
// Test Case 6: String and char literals
#include <stdio.h>

int main() {
    char* s1 = "Hello World";
    char* s2 = "Hello \"Quoted\" World";
    char* s3 = "Line1\nLine2";
    char c1 = 'a';
    char c2 = '\n';
    char c3 = '\'';

    printf("%s %c\n", s1, c1);
    return 0;
}
```

7.7 Test Case 7: Custom Functions with Parameters and Calls

This test case checks the scanner's ability to track function definitions, parameters, and calls.

```
// Test Case: Custom functions with parameters and calls
#include <stdio.h>

// Simple addition function
int add(int a, int b) {
    return a + b;
}

// Function with multiple parameters and local variables
float calculate(int x, float y, char op) {
    float result = 0.0;

    if (op == '+') {
        result = x + y;
    } else if (op == '-') {
        result = x - y;
    } else if (op == '*') {
        result = x * y;
    } else if (op == '/') {
        if (y != 0) {
            result = x / y;
        }
    }

    return result;
}

// Main function calling the custom functions
int main() {
    int sum = add(5, 10);
    printf("Sum: %d\n", sum);

    float calc_result = calculate(20, 5.5, '+');
    printf("Calculation result: %.2f\n", calc_result);

    // Testing another operation
    calc_result = calculate(20, 5.5, '*');
    printf("Calculation result: %.2f\n", calc_result);

    return 0;
}
```

7.8 Test Case 8: Improved Function Tests

This test case tests function declarations with various parameter types and function calls.

```
// Test Case: Improved function tests
#include <stdio.h>
```

```
// Function with parameters and return value
int multiply(int x, int y) {
    return x * y;
}

// Function with no parameters
void printMessage() {
    printf("Hello from function!\n");
}

// Function with array parameter
int sumArray(int arr[], int size) {
    int total = 0;
    for (int i = 0; i < size; i++) {
        total += arr[i];
    }
    return total;
}

int main() {
    // Simple function call with integer constants
    int product = multiply(6, 7);

    // Function call with variables
    int a = 10;
    int b = 5;
    int result = multiply(a, b);

    // Function with no arguments
    printMessage();

    // Function with array
    int numbers[] = {1, 2, 3, 4, 5};
    int sum = sumArray(numbers, 5);

    printf("Results: %d, %d, %d\n", product, result, sum);

    return 0;
}
```

8 Handling of Comments, Strings, and Errors

8.1 Comment Handling

The scanner supports two types of comments:

1. **Single-line comments** (`// comment`): Matched by the pattern `//.*/` and simply

ignored.

2. **Multi-line comments** (`/* comment */`): Handled using a special lexical state `COMMENT`. When `/*` is encountered, the scanner enters the `COMMENT` state and ignores all input until it finds `*/`.

Nested comments are partially supported. The scanner will exit comment mode upon finding the first `*/`, potentially leading to syntax errors if nested comments are used improperly.

8.2 String Handling

String literals are processed using a dedicated `STRING` state:

1. When a double quote is encountered, the scanner enters the `STRING` state.
2. It then collects all characters until a closing double quote is found.
3. Escape sequences within strings (like `\n`, `\"`) are properly recognized.
4. Unterminated strings (without closing quotes) generate an error message.

The scanner also supports continued strings across multiple lines using backslash-newline sequences.

8.3 Error Handling

The scanner implements several error detection mechanisms:

1. **Invalid tokens**: Characters that don't match any defined pattern are reported as errors with line numbers.
2. **Malformed numbers**: Invalid numeric formats (like `20.5.3`) are flagged as errors.
3. **Unterminated strings/comments**: The scanner detects and reports strings without closing quotes.

Errors are reported to stderr with the line number and the invalid token:

```
[line 4] ERROR: Invalid token '$'
```

The scanner continues processing after encountering errors, allowing it to find multiple errors in a single pass.

9 Conclusion

Our lexical analyzer for the C programming language successfully meets all the requirements for a robust scanner. It correctly identifies all C language tokens, maintains detailed symbol and constant tables, tracks context for declarations, assignments, and function calls, reports errors for invalid tokens, and handles complex constructs like strings, comments, and function arguments.

The implementation uses efficient data structures and algorithms, particularly an array-based symbol table that provides straightforward implementation and maintenance. The scanner's modular design and clear documentation make it maintainable and extendable for future enhancements.

Recent improvements, such as associating constants with their variable names, provide additional functionality beyond the basic requirements of a lexical analyzer. These features make our scanner not just a token recognizer but also a valuable tool for gathering information that would be useful in subsequent compilation phases.

10 References

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley.

2. Levine, J., Mason, T., & Brown, D. (1992). Lex & Yacc (2nd Edition). O'Reilly Media.

3. Flex Manual: https://westes.github.io/flex/manual/

4. C Language Specification: https://www.open-std.org/jtc1/sc22/wg14/