[Robert D. Carswell]

[23 January 2024]

project 1 report

CMSC 430 – COMPILER THEORY AND DESIGN

Contents

[Executive Summary 2](#_Toc156910714)

[Testing 3](#_Toc156910715)

[Test Case Table 3](#_Toc156910716)

[Test Case Screenshots 4](#_Toc156910717)

[Test Case 1 4](#_Toc156910718)

[Test Case 2 5](#_Toc156910719)

[Test Case 3 5](#_Toc156910720)

[Test Case 4 6](#_Toc156910721)

[Test Case 5 6](#_Toc156910722)

[Test Case 6 7](#_Toc156910723)

[Test Case 7 7](#_Toc156910724)

[Test Case 8 8](#_Toc156910725)

[Approach 9](#_Toc156910726)

[Lessons Learned 9](#_Toc156910727)

# Executive Summary

The project focuses on enhancing the existing lexical analyzer and compilation listing generator code, explicitly targeting the scanner.l file. The modifications include adding new language features, including reserved words such as else, elsif, endfold, endif, fold, if, left, real, right, and then. Each reserved word becomes a distinct token represented in all uppercase. Two logical operators, | (OROP) and ! (NOTOP), as well as five relational operators (=, <>, >, >=, and <=) represented by the single token RELOP, are introduced. The subtraction operator (-) becomes a new lexeme for the ADDOP token, while the division operator (/) is added for the MULOP token. Additionally, new tokens include REMOP (remainder operator, %), EXPOP (exponentiation operator, ^), and NEGOP (unary minus operator, ~). A new type of comment, starting with -- and ending with the end of the line, is introduced. Identifier definitions are modified to allow underscores, with restrictions on consecutive underscores. Two new types of literals are added: hexadecimal integers (starting with # followed by digits or A-F) and real literals (allowing digits, decimal point, and an optional exponent). Character literals are modified to permit five additional escape characters: '\b', '\t', '\n', '\b', and '\f'. Additionally, the tokens.h header file is updated to include definitions for each new token. These major changes greatly increase the lexical analyzer's abilities, making them more flexible to support a wider range of language structures and improving the accuracy of the compilation listing generator.

# Testing

## Test Case Table

| Test Cases | Description | Expected Output | Actual Output | Pass/Fail |
| --- | --- | --- | --- | --- |
| Test Case 1 | Read test file 1 | File contents with :  Msg (Bottom of file):  Compilation Successful | File contents with :  Msg (Bottom of file):  Compilation Successful | Pass |
| Test Case 2 | Read test file 2 | File contents with:  Msg (Under line 5):  Lexical Error,  Invalid Character $  Msg (Bottom of file): Lexical Errors: 1  Syntax Errors: 0  Semantic Errors: 0 | File contents with:  Msg (Under line 5):  Lexical Error,  Invalid Character $  Msg (Bottom of file): Lexical Errors: 1  Syntax Errors: 0  Semantic Errors: 0 | Pass |
| Test Case 3 | Read test file 3 | File contents with :  Msg (Bottom of file):  Compilation Successful | File contents with :  Msg (Bottom of file):  Compilation Successful | Pass |
| Test Case 3 | Read test file 4 | File contents with :  Msg (Bottom of file):  Compilation Successful | File contents with :  Msg (Bottom of file):  Compilation Successful | Pass |
| Test Case 3 | Read test file 5 | File contents with :  Msg (Bottom of file):  Compilation Successful | File contents with :  Msg (Bottom of file):  Compilation Successful | Pass |
| Test Case 3 | Read test file 6 | File contents with :  Msg (Bottom of file):  Compilation Successful | File contents with :  Msg (Bottom of file):  Compilation Successful | Pass |
| Test Case 3 | Read test file 7 | File contents with:  Msg (Under line 5):  Lexical Error,  Invalid Character $  Msg (Under line 5):  Lexical Error,  Invalid Character ?  Msg (Bottom of file): Lexical Errors: 2  Syntax Errors: 0  Semantic Errors: 0 | File contents with:  Msg (Under line 5):  Lexical Error,  Invalid Character $  Msg (Under line 5):  Lexical Error,  Invalid Character ?  Msg (Bottom of file): Lexical Errors: 2  Syntax Errors: 0  Semantic Errors: 0 | Pass |
| Test Case 3 | Read test file 6 | File contents with:  Msg (Under line 12):  Lexical Error,  Invalid Character \_  Msg (Under line 12):  Lexical Error,  Invalid Character \_  Msg (Under line 12):  Lexical Error,  Invalid Character \_  Msg (Under line 13):  Lexical Error,  Invalid Character \_  Msg (Under line 14):  Lexical Error,  Invalid Character \_  Msg (Bottom of file): Lexical Errors: 5  Syntax Errors: 0  Semantic Errors: 0 | File contents with:  Msg (Under line 12):  Lexical Error,  Invalid Character \_  Msg (Under line 12):  Lexical Error,  Invalid Character \_  Msg (Under line 12):  Lexical Error,  Invalid Character \_  Msg (Under line 13):  Lexical Error,  Invalid Character \_  Msg (Under line 14):  Lexical Error,  Invalid Character \_  Msg (Bottom of file): Lexical Errors: 5  Syntax Errors: 0  Semantic Errors: 0 | Pass |

# Test Case Screenshots

### Test Case 1

A black screen with white text

Description automatically generated

### Test Case 2

A computer screen shot of a black screen

Description automatically generated

### Test Case 3

A screenshot of a computer

Description automatically generated

### Test Case 4

A screenshot of a computer program

Description automatically generated

### Test Case 5

A screenshot of a computer program

Description automatically generated

### Test Case 6

A screenshot of a computer

Description automatically generated

### Test Case 7

A screen shot of a computer

Description automatically generated

### Test Case 8

A screenshot of a computer

Description automatically generated

# Approach

In building upon the foundational knowledge acquired during week one, which primarily covered the general structure of the program and the interaction between its components, I focused on implementing changes according to requirements 1–8 as outlined in the instructions. For requirements 9–13, a shift in perspective was essential, recognizing the potential unintended outcomes of regular expressions with even minor alterations. Adopting a literal approach using the expression ('.'|'\\b'|'\\t'|'\\n'|'\\b'|'\\f'), I ensured that the program only accepted the specific escape characters requested. This meticulous approach underscores the necessity for clear rules and standards in software development, especially when handling literals. The lack of well-established guidelines risks users not achieving their intended outcomes. Therefore, continuous review and validation throughout the software development lifecycle become imperative to align the work with project objectives, emphasizing a commitment to both consistency and quality.

# Lessons Learned

The literal approach vividly illustrates the consequences of lacking firm rules and standards, as users may not obtain their desired outcomes, particularly in requirements 9–13. Regular expressions were employed to formalize acceptance parameters, acknowledging that misunderstanding the requirements could yield unintended results. Requirement 13 mandated the modification of character literals with an existing '.' to accept five additional escape characters using a '\f' format. A seemingly straightforward solution of escaping the backslash ('\\.') led to complexities, as characters previously accepted, like 'a,' would no longer be valid, potentially causing errors. Such straightforward misunderstandings highlight the potential difficulties in the complex environment where compilers operate, constrained by rules governing their behavior to produce expected results consistently.However, the approach document provided was more than helpful and kept me within the scope of the requirements.