Philip M. Seisman

CMSC 430 – Compiler Theory and Design

Professor George Mikulski

January 21, 2025

**Project 1**

**Approach**

The primary objective of this project was to modify the lexical analyzer (scanner.l) and the compilation listing generator code to meet specific requirements. The modifications involved adding new tokens, reserved words, operators, and literals, as well as refining error management to provide a detailed summary of errors.

Steps Taken:

1. Adding Reserved Words:
   * Reserved words else, elsif, endfold, endif, fold, if, left, real, right, and then were added as separate tokens.
   * Token names were created in uppercase to match the lexemes (e.g., ELSE for else).
2. Adding Logical Operators:
   * Added two logical operators:
     + | with the token OROP.
     + ! with the token NOTOP.
3. Adding Relational Operators:
   * Five relational operators were added: =, <>, >, >=, and <=.
   * All were represented by the token RELOP.
4. Adding Arithmetic Operators:
   * Added - as a lexeme for the ADDOP token.
   * Added / as a lexeme for the MULOP token.
   * Introduced new tokens for additional operators:
     + REMOP for % (remainder operator).
     + EXPOP for ^ (exponentiation operator).
     + NEGOP for ~ (unary minus operator).
5. Enhancing Comments:
   * Added support for a second type of comment starting with -- and ending at the end of the line. No token is returned for these comments.
6. Modifying Identifier Rules:
   * Updated the id rule to allow underscores but restricted them as follows:
     + No leading or trailing underscores.
     + No more than two consecutive underscores.
7. Adding Hexadecimal Integers:
   * Introduced a rule for hexadecimal integers, beginning with # followed by one or more hexadecimal digits (0-9, A-F, a-f).
8. Adding Real Literals:
   * Added real literals starting with a sequence of digits, a mandatory decimal point, and one or more digits after the decimal point.
   * Allowed an optional exponent (e or E followed by an optional + or - and digits).
9. Enhancing Character Literals:
   * Allowed additional escape characters in character literals: \b, \t, \n, \r, and \f.
10. Updating Tokens Header (tokens.h):
    * Added new tokens (OROP, NOTOP, RELOP, REMOP, EXPOP, and NEGOP).
11. Updating Error Management:
    * Modified the following functions in listing.cc:
      + lastLine: Displays the total number of errors (lexical, syntactic, semantic) or a success message if no errors occurred.
      + appendError: Counts errors by category and queues error messages.
      + displayErrors: Displays all error messages for the current line and clears the queue.

**Test Cases**

**Test Case 1:** First build the skeleton for project 1 as shown in part 5 of the video series on lexical analysis using the make file provided. Then run it on the test cases test1.txt – test3.txt that are provided in Project 1 Test Data and be sure that you understand how it works. Examine the contents of lexemes.txt, so that you see the lexeme-token pairs that it contains.

A computer screen with white text

Description automatically generated

A computer screen with white text

Description automatically generated

A screenshot of a computer

Description automatically generated

**Test Case 2:** A good starting point would be item 1 in the requirements, which includes the additional reserved words of the language. Each of these is a separate token and requires a separate translation rule. Examine the existing translation rules for the reserved words as an example of how to proceed. In addition, add the token names for each one to the enumerated type Tokens in tokens.h. The order in which you add them is unimportant. Rebuild the program with the make file to ensure that it builds correctly.

**A screenshot of a computer program

Description automatically generated**

**Test Case 3:** Adding all the operators as specified by items 2-8 in the requirements would be a good next step. Examine the existing translation rules for the existing operators as an example of how to proceed. As before, you must also add the token names for each new operator to the enumerated type Tokens in tokens.h.

**A screenshot of a computer

Description automatically generated**

**Test Case 4:** As the last modification to scanner.l and tokens.h, add the new comment, modify the identifier and character literal tokens and add the real literal, and hexadecimal integer literal tokens as specified by items 9-13 in the requirements.

**A computer screen shot of a black screen

Description automatically generated**

**Test Case 5:** The final required change is to modify the three functions that generate the compilation listing as described in the requirements so that the number of errors are displayed at the end if any occur, or the message Compilation Successful is displayed if none occur. In addition, the modifications should ensure that all the error messages that have occurred on the previous line are displayed. Rerunning any of the previous test cases should confirm that the Compilation Successful is displayed.

A screenshot of a computer error

Description automatically generated

**Test Case 6:** As a final test, test8.txt contains every punctuation symbol, reserved word, operator and both valid and invalid identifiers. Shown below is the output that should result when using that test case as input:

**A screenshot of a computer program

Description automatically generated**

**A black background with blue text

Description automatically generated**

**Lessons Learned and Improvements**

While I was working on this project, it really provided valuable insights into the complexity of creating a lexical analyzer. Crafting regular expressions to meet specific requirements, such as restricting consecutive underscores and handling hexadecimal integers, required careful planning and iterative testing. Implementing a comprehensive error management system demonstrated the importance of clear error categorization and reporting to aid debugging. Additionally, thorough testing of various edge cases proved crucial in ensuring the program's reliability and adherence to the requirements.

**Potential Improvements**

One example of a potential improvement it to include line numbers in error reports that could significantly improve the user experience by making it easier to locate issues. Optimizing some of the regular expressions could make the code more efficient and readable. Extending the analyzer to handle additional constructs, such as multi-line comments and string literals, would enhance its functionality. Also, providing categorized error summaries and suggestions for resolving issues would offer a more user-friendly experience.