# **C-Minus Scanner Project-3 Report**

## **COMSC 440: Language Translation and Compiler Design**

**Spring 2025**

**Group Members: Matt Dugal, Luca Martinez, Louis Sader  
Project 3: C-Minus Scanner  
Due Date: March 7, 2025**

### **1. Introduction**

#### **Overview of the Project and Its Objectives**

The objective of this project is to develop a scanner for the C-Minus (C-) programming language. The scanner is a crucial component of the compiler, which is responsible for recognizing / categorizing tokens. The project involves implementing a Deterministic Finite Automaton (DFA) to identify keywords, special symbols, identifiers, numeric values, white spaces, and comments. By using the TINY compiler as a template, our team aims to create a scanner-only version of a C- compiler.

#### **Description of the C-Minus Language**

C-Minus (C-) is a subset of the C programming language that includes programming constructs such as integer variables, arrays, functions, local and global declarations, conditional statements (if-else), and loops (while). Execution of a C- program starts with a call to the main function, which must be declared last. The lexical conventions of C- define the valid tokens recognized by the scanner, which include:

* **Keywords**: if, else, int, return, void, while
* **Special Symbols**: +, -, *, /, <, <=, >, >=, ==, !=, =, ;, ,, (, ), [, ], {, }, /* \*/
* **Identifiers (ID)**: A sequence of letters beginning with an alphabetic character
* **Numbers (NUM)**: A sequence of digits
* **Whitespace**: Spaces, newlines, and tab characters
* **Comments**: Enclosed between /\* and \*/

The scanner must accurately recognize these tokens while ignoring irrelevant whitespace and correctly handling comments.

**Purpose of the Scanner in the Compilation Process**

The scanner, also known as the lexical analyzer, serves as the first phase of the compilation process. It reads the source code and converts it into a sequence of tokens that the parser can process. The scanner eliminates unnecessary characters such as whitespace and comments while ensuring that the input is structured according to the defined lexical rules. By using a DFA, the scanner efficiently identifies token boundaries and classifies them based on predefined categories.

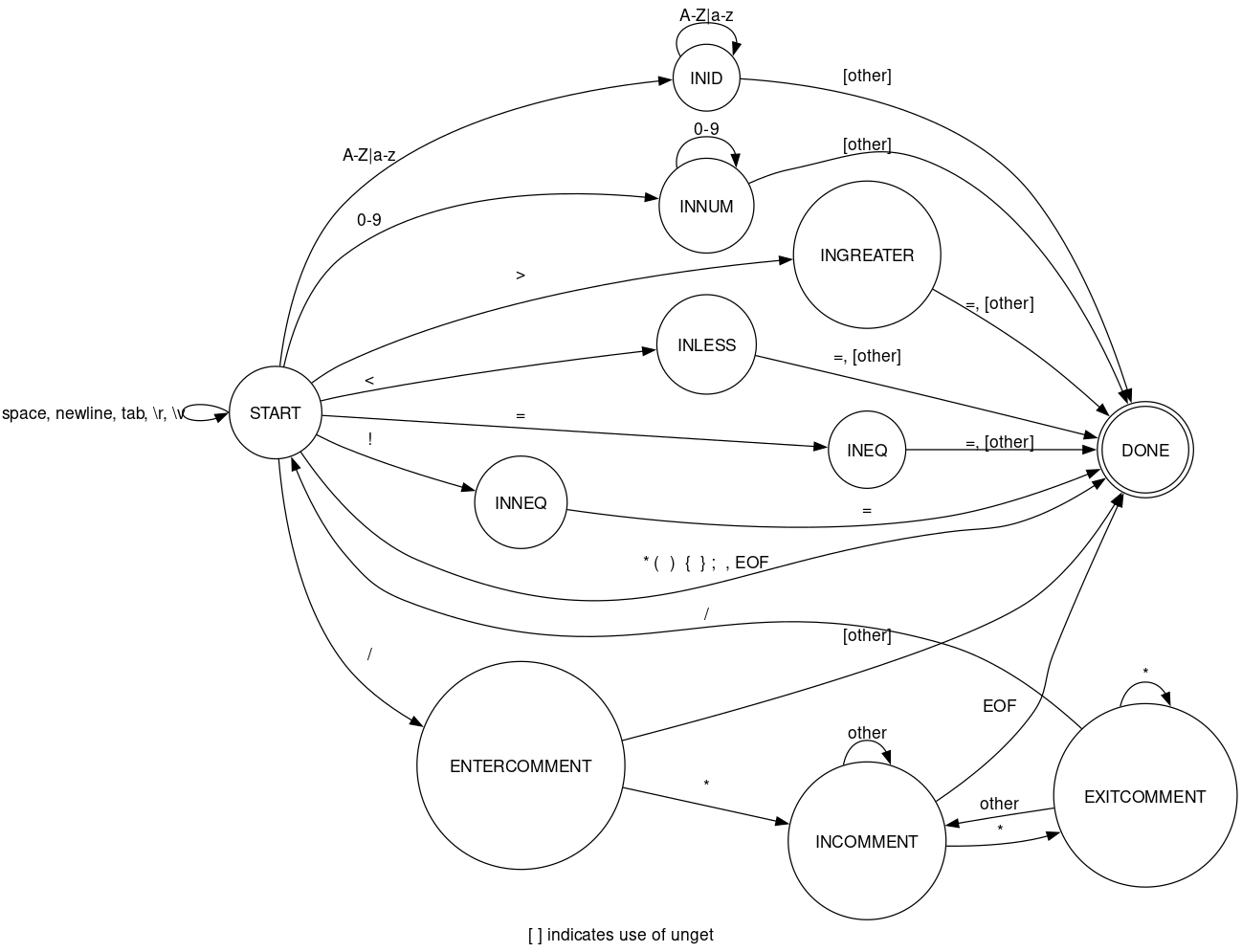
#### **Team Member Contributions and Roles**

To effectively divide the workload, each team member focused on specific aspects of the project:

* **Matt Dugal- DFA and gcd.cm**: Responsible for designing and documenting the DFA used for token recognition. Created a state transition diagram and ensured accurate representation of C- lexical rules. Implemented the gcd.cm program to serve as a test case for the scanner. Ensured correct formatting and syntax based on C- conventions.
* **Louis Sader/Matt Dugal- scan.c**: Developed the scan.c file to implement the scanning logic. Incorporated binary search for reserved word lookup and ensured the correct identification of tokens.
* **Luca Martinez- Makefile**: Designed the Makefile to compile and link necessary files. Simplified the build process by removing unnecessary components from the original TINY compiler Makefile.

### **2. DFA Design and Implementation**

*Responsible: Matt Dugal*



### **3. Implementation of scan.c**

*Responsible: Louis Sader/ Matt Dugal*

### **Token Recognition**

### The scanner uses a deterministic finite automaton (DFA) to traverse different states based on input characters. The recognized token types include:

* **Identifiers (ID)**: Names of variables and functions.
* **Keywords**: Reserved words such as if, else, while, return, etc.
* **Operators and Symbols**: Includes arithmetic operators (+, -, \*, /), relational operators (<, >, <=, >=, ==, !=), and delimiters (;, ,, {, }, [, ], (, )).
* **Numbers (NUM)**: Integer literals.
* **Comments**: The scanner correctly skips block comments (/\* ... \*/).

#### 

#### **Handling Reserved Words**

The scanner implements a binary search to differentiate between identifiers / reserved words. Reserved words are stored in an array of structures, which contain the keywords and their corresponding token type. When an identifier is encountered, a binary search is performed to check if it matches any reserved word.

#### **Binary Search Optimization**

To optimize token recognition, the ReservedLookupB() function implements a recursive binary search to match a given string against reserved keywords. This ensures an average time complexity of O(log n), which is an improvement over the O(n) complexity of a linear search.

### **How the DFA Was Translated into Code**

The DFA is implemented using a state-based approach, where an enum called StateType defines the possible states:

typedef enum {

START, INLESS, INEQ, INGREATER, INNEQ, INNUM, INID, INCOMMENT, DONE

} StateType;

The scanner transitions between these states based on character input. The primary function getToken() processes input characters and determines the appropriate token by moving between states. Key transitions include:

* Transitioning to INID for alphabetic characters (potential identifiers or reserved words).
* Transitioning to INNUM for numeric characters.
* Handling multi-character operators (<=, >=, !=, ==).
* Skipping whitespace and comments.

**Challenges Faced and Solutions**

#### 1. Handling Block Comments (/\* ... \*/) Properly

* **Problem**: Problem with making sure that the multi-line comments are ignored until the closing \*/.
* **Solution**: Implemented a state transition to INCOMMENT, consuming characters until \*/ is detected, preventing tokenization of comment content.

#### 2. tokenNames Error

* **Problem**: tokenNames was never declared and it wasn’t in the tiny scanner so errors were being thrown.
* **Solution**: Simply remove it.

#### 3. Having duplicated outputs for ID and Numbers

* **Problem**: While the majority of the scanner was correct at the end stage, it was duplicating each character of id and num tokens.
* **Solution**: Removing duplicated statements in scanner.

### **4. gcd.cm Implementation**

*Responsible: Matt Dugal*

* Was a basic implementation of Euclid’s algorithm in C- copied from the reference section from Louden’s book.
* Purpose is to test the scanner output using a known good piece of code.

### **5. Makefile Implementation**

*Responsible: Luca Martinez*

* The purpose of this Makefile is to automate the build process, which is done by defining the rules for compiling, linking, and managing dependencies. This ensures that only necessary parts get compiled after changes are made. This was all done by editing the original Makefile from a previous project. One challenge faced was figuring out all the errors after creating the Makefile. We only found out about them because we ran the whole test as one. The main error we found through this is a duplicate error in the scan.c file. Which one a few of these duplicates were removed the file ran smoothly.

### **6. Testing and Results**

* How to Test the Makefile - Tests the whole program
  + make
    - To ensure you successfully compiled the entire program
  + ./cminus gcd.cm
    - This should output the list given in the project specifications.

### Example Token Output from a Test Run

* + Given the following C-Minus source code snippet:

int main() {

if (x >= 10) return x;

}

The scanner would produce the following tokens:

Token: INT Lexeme: int

Token: ID Lexeme: main

Token: LPAREN Lexeme: (

Token: RPAREN Lexeme: )

Token: LBRACE Lexeme: {

Token: IF Lexeme: if

Token: LPAREN Lexeme: (

Token: ID Lexeme: x

Token: GE Lexeme: >=

Token: NUM Lexeme: 10

Token: RPAREN Lexeme: )

Token: RETURN Lexeme: return

Token: ID Lexeme: x

Token: SEMI Lexeme: ;

Token: RBRACE Lexeme: }

**Sample output from the scanner when run on gcd.cm**

vboxuser@Ubuntu:~/Project 3$ ./cminus gcd.cm

>>>>> C-MINUS COMPILATION: gcd.cm

1: int gcd(a, b){

1: reserved word: int

1: ID, name = gcd

1: (

1: ID, name = a

1: ,

1: ID, name = b

1: )

1: {

2: if(b==0){

2: reserved word: if

2: (

2: ID, name = b

2: ==

2: NUM, val = 0

2: )

2: {

3: return a;

3: reserved word: return

3: ID, name = a

3: ;

4: }

4: }

5: else{

5: reserved word: else

5: {

6: return gcd(b. a-a/b\*b);

6: reserved word: return

6: ID, name = gcd

6: (

6: ID, name = b

6: ERROR: .

6: ID, name = a

6: -

6: ID, name = a

6: /

6: ID, name = b

6: \*

6: ID, name = b

6: )

6: ;

7: }

7: }

8: }

8: }

9:

10: void main(void){

10: reserved word: void

10: ID, name = main

10: (

10: reserved word: void

10: )

10: {

11: int a = input();

11: reserved word: int

11: ID, name = a

11: =

11: ID, name = input

11: (

11: )

11: ;

12: int b = input();

12: reserved word: int

12: ID, name = b

12: =

12: ID, name = input

12: (

12: )

12: ;

13: output(gcd(a, b));

13: ID, name = output

13: (

13: ID, name = gcd

13: (

13: ID, name = a

13: ,

13: ID, name = b

13: )

13: )

13: ;

14: } 14: }

15: EOF

### 