Universitat Pompeu Fabra Barcelona

Aranich, Marcel - 251453
Quera, Clàudia - 231197
Villarino, Jorge - 231351
Solans, Arnau - 216530
Garcia, David - 251587
Prat, Ariadna - 251281

Project Overview

In this lab project, we aim to develop a C Scanner tool that facilitates lexical analysis of the C code.

A C Scanner is a fundamental tool in software development, as it plays a crucial role in the compilation process by breaking down the source code into tokens, which are the building blocks of syntax analysis and interpretation. The Scanner improves code readability, enables syntax highlighting, and provides essential information for subsequent compilation phases

Project Goals and Objectives

The primary goal of this project is to create a reliable C Scanner with comprehensive tokenization capabilities. The objectives include:

- Developing a flexible and efficient scanner capable of recognizing various token categories, including identifiers, keywords, numbers, literals, special characters, and operands.
- Implementing some deterministic finite automatons (DFAs) to handle token recognition based on predefined patterns and rules.
- Designing a well-structured codebase which works correctly for all the specifications described in the lab statement.

Architecture and Design of the code

In order to improve modularity and maintainability, we have divided the codebase into separate files:

1. main.c:
    - Acts as the entry point for the program.
    - Opens, creates, prepares and processes the input and output files.
    - Orchestrates the overall workflow of the scanner.

2. datastructures.c:
    - Defines and implements the data structures used in the scanner, including both the Token and the DFA structures.

- Defines also the functions which will work with the DFAs.
- Ensures proper memory management by deallocating the memory allocated for DFA components.

3. debug.c:
    - Provides essential functionality for debugging the token processing during the execution of the program.

4. errors.c:
    - Defines error codes and variables in the form of messages for error handling through all the codebase.
    - Helps in managing and communicating the errors that occur during the execution.

5. utils.c:
    - Holds utility functions shared across different components.
    - Encapsulates common functionalities, promoting code reuse and reducing redundancy.
    - Enhances overall code readability and maintainability.

6. dfatables.h:
    - Defines the structure and behavior of the DFAs used for the lexical analysis in the program.
    - Defines and creates all the types, identifiers, keywords, operators, literals and special characters (and their respective tables) which are part of the alphabet of our program.

// Every .c file named here includes its respective .h file, which ends by having the same functionalities as its respective .c file.

This modular file structure promotes clarity, maintainability, and collaborative development. Team members can work separately on some functionalities thanks to this distribution, which promotes teamwork during the development process. However, in this project, as some code files were much shorter than the others, all of us have worked in almost every code file, helping each other, whe. someone was stuck in any process.

Error Handling

In managing errors, I've adopted a structured approach by defining error codes and their corresponding messages in a separate errors.h header file. This allows for clear organization and separation of concerns within the codebase. By keeping error-related logic isolated, it enhances readability and maintainability, making it easier to locate and update error handling logic as needed. Furthermore, this approach promotes consistency across the codebase, ensuring that error codes and messages remain uniform throughout the program. It also facilitates future scalability, as adding new error codes and messages can be seamlessly integrated by updating the errors.h file.

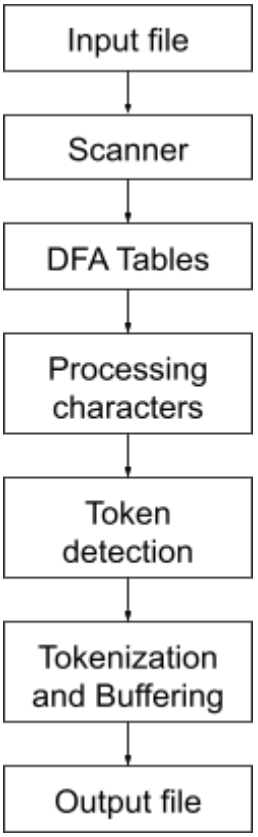
Alphabet definition

Following the specification of the lexemes described in the statement, we have derived this alphabet:

Σ = { **a-z**, **A-Z**, **0-9**, ‘ ‘, ‘**,**’, ‘**;**’, ‘**(**‘, ‘**)**’, ‘**[**‘, ‘**]**’, ‘**{**‘, ‘**}**’, =, +, >, *, ‘ “ ‘}

Overview of how the scanner works

The following flowchart represents the functioning of our program:



1. Initialization:

- The scanner begins by opening the input file provided as a command-line argument. It also creates an output file to store the tokenized contents.

2. DFA Initialization:

- Several Deterministic Finite Automata (DFA) structures are initialized, each corresponding to different token types such as types, identifiers, keywords, numbers, special characters, operators, and literals.
- These DFAs are constructed with transition tables and accepting states, which help recognize patterns corresponding to different token types.

3. Scanning Process:

- The scanner reads the input file character by character.
- As it reads characters, it updates the DFAs with the current character to recognize patterns and transition to new states.
- Whenever it encounters a delimiter (=*{' ', EOF, \n, \r } U Operators U SpecialCharacters = {' ', EOF, \n, \r, *, +, =, >, (,), ;, {, }, [,], ','} + '\0'*), it finalizes the current token being processed by the DFAs.
- If the DFAs successfully recognize a token, it invokes a function to process and tokenize the recognized string based on the DFA that matched.
- If a token cannot be recognized by any DFA, it is handled as an error token.

4. Buffering and Writing to Output:

- As tokens are recognized and processed, the scanner writes them into a writing buffer.
- The writing buffer is flushed into the output file when it reaches a certain threshold or when the end of the file is reached.
- This process continues until the entire input file is processed.

5. Debug Mode:

- When the token is built, we apply the Debug Mode if the variable debug is DEBUG_ON. In this case, the tokens are ordered in their respective line.
- When the function debug_mode detects an element "\n", the number increases and the text tokens are printed to the next line.
- In the case where debug_mode is DEBUG_OFF, we print the normal format, without tokens ordered in lines.

6. Cleanup:

- After processing is complete, the scanner frees memory allocated for various buffers and closes the input and output files.

7. Error Handling:

- The scanner handles errors such as file opening errors, memory allocation errors, and unrecognized tokens.

8. Main Function:
   - The `main` function handles command-line arguments, calls the `processFile` function, and checks for any errors returned by the processing function.

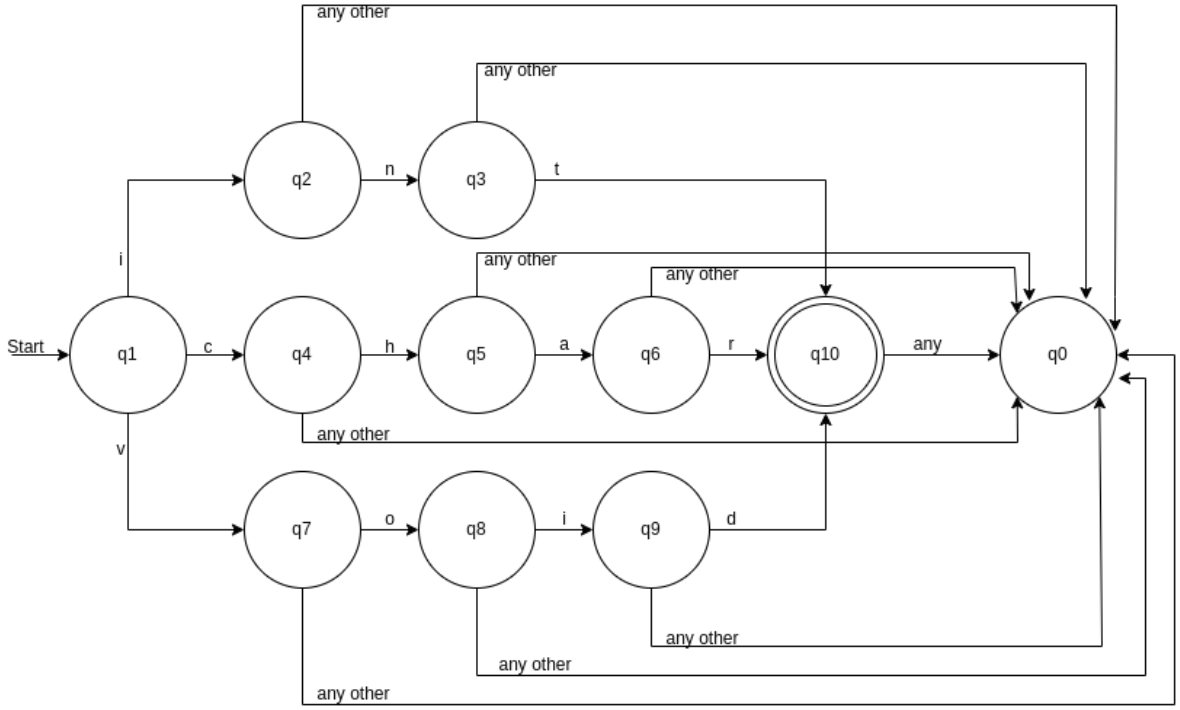
DFA definitions

DFA for types:

- Accepts “int”, “char” and “void”:

	i	**n**	**t**	**c**	**h**	**a**	**r**	**v**	**o**	**d**	**other**
Reject (q0)	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
Start (q1)	q2	q0	q0	q4	q0	q0	q0	q7	q0	q0	q0
q2	q0	q3	q0	q0	q0	q0	q0	q0	q0	q0	q0
q3	q0	q0	q10	q0	q0	q0	q0	q0	q0	q0	q0
q4	q0	q0	q0	q0	q5	q0	q0	q0	q0	q0	q0
q5	q0	q0	q0	q0	q0	q6	q0	q0	q0	q0	q0
q6	q0	q0	q0	q0	q0	q0	q10	q0	q0	q0	q0
q7	q0	q0	q0	q0	q0	q0	q0	q0	q8	q0	q0
q8	q9	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
q9	q0	q0	q0	q0	q0	q0	q0	q0	q0	q10	q0
q10*	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0

‘other’ maps every item of our alphabet differently from the previous columns.



DFA for identifiers

Rules for identifiers: only letters (both uppercase and lowercase) and numbers. Underscore (‘_’) is not allowed since it's not part of the alphabet. Also, the first character must NOT be a digit.

	Number	**Lowercase**	**Uppercase**	**other**
Reject (q0)	q0	q0	q0	q0
Start (q1)	q0	q2	q2	q0
q2*	q2	q2	q2	q0

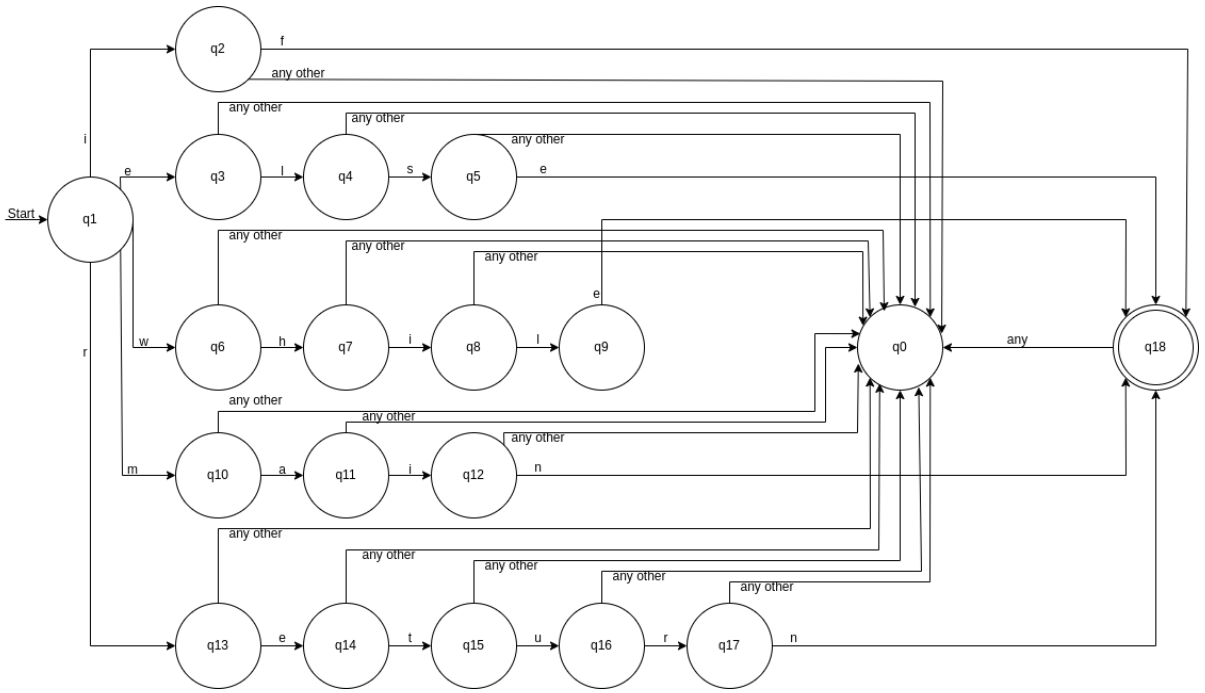
Number maps {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, Lowercase maps {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}, Uppercase maps {A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z} and ‘other’ maps every item of our alphabet not mapped by the previous columns.

DFA Keywords: if, else, while, main, return

The starting state is q0, and the accepting state is q_f. When the current state is one of the previous states, the automata ends, and the string is accepted and kept in a buffer. The state q_n is the rejecting state, when we are in this state, we reject the string. In this automata, we only accept the strings “if, else, while, main, return”. This is the alphabet “i, f, e, l, s, w, h, m, a, n, r, t, u, other”.

	i	**f**	**e**	**l**	**s**	**w**	**h**	**m**	**a**	**n**	**r**	**t**	**u**	**other**
Reject (q0)	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
Start (q1)	q2	q0	q3	q0	q0	q6	q0	q10	q0	q0	q13	q0	q0	q0
q2	q0	q18	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
q3	q0	q0	q0	q4	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
q4	q0	q0	q0	q0	q5	q0	q0	q0	q0	q0	q0	q0	q0	q0
q5	q0	q0	q18	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
q6	q0	q0	q0	q0	q0	q0	q7	q0	q0	q0	q0	q0	q0	q0
q7	q8	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
q8	q0	q0	q0	q9	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
q9	q0	q0	q18	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
q10	q0	q0	q0	q0	q0	q0	q0	q0	q11	q0	q0	q0	q0	q0
q11	q12	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
q12	q0	q0	q0	q0	q0	q0	q0	q0	q0	q18	q0	q0	q0	q0
q13	q0	q0	q14	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0
q14	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q15	q0	q0
q15	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q16	q0
q16	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q17	q0	q0	q0
q17	q0	q0	q0	q0	q0	q0	q0	q0	q0	q18	q0	q0	q0	q0
q18*	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0	q0

‘other’ maps every item of our alphabet differently from the previous columns.



DFA for numbers {0-9}*:

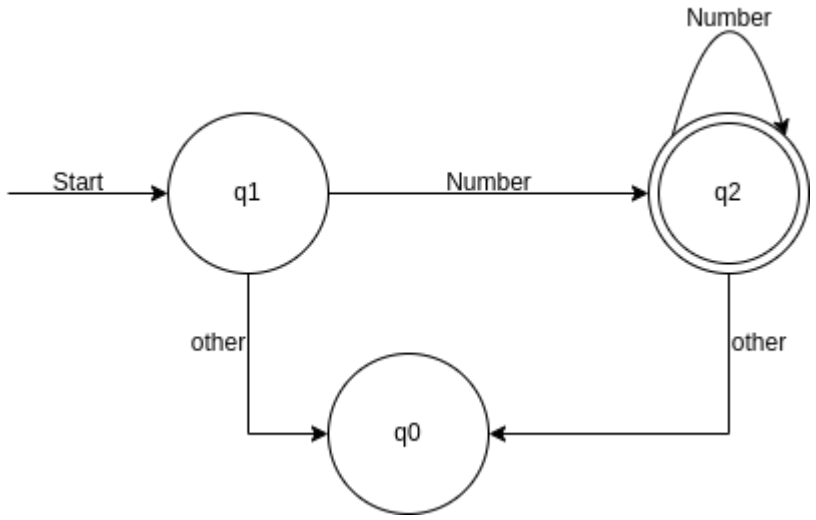
q0 represents the starting state, any input being a number from 0 to 9 leads to state q1, any other input leads to state qn.

State qn is the rejecting state, indicating that the input read is not a number if the automata reaches this state, any input received keeps the automata in this state.

State q1 is the accepting state, it keeps looping itself when the input is a number between 0 to 9, if any other input is received it goes to state qn.

	Number	**other**
Reject (q0)	q0	q0
Start (q1)	q2	q0
q2*	q2	q0

Number maps {0,1,2,3,4,5,6,7,8,9} and ‘other’ maps every item of our alphabet different from the previous column.



DFA for special characters (“;”, “,”, “[“, “]”, “{“ , “}”, “(“, “)”):

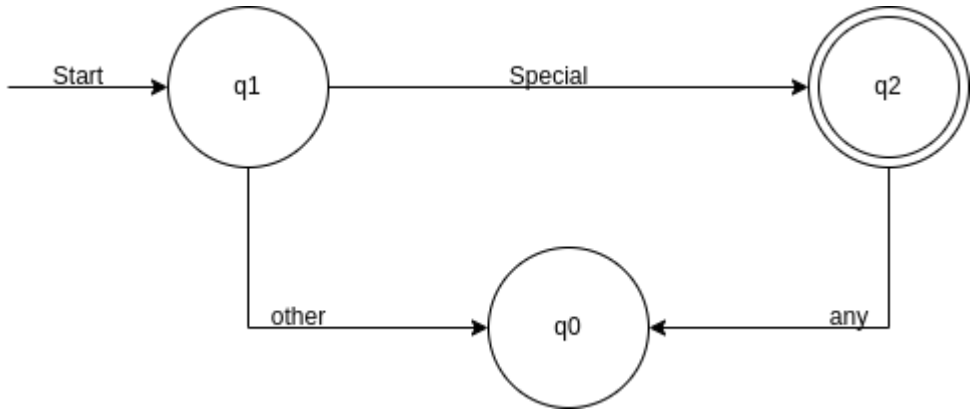
q0 represents the starting state, any input of “,”, “;”, “(“, “)”, “[“, “]”, “{“, “}” leads to state q1, any other input to state qn.

q1 is the accepting state, any input received kicks the automata out of this state to state qn.

qn is the rejecting state, indicating that the input read is not a special character, any input keeps the automata in this state.

	Special	**other**
Reject (q0)	q0	q0
Start (q1)	q2	q0
q2*	q0	q0

Special maps {**,** , **;** , **(** , **)** , **[** , **]** , **{** , **}** } and ‘other’ maps every item of our alphabet different from the previous column.



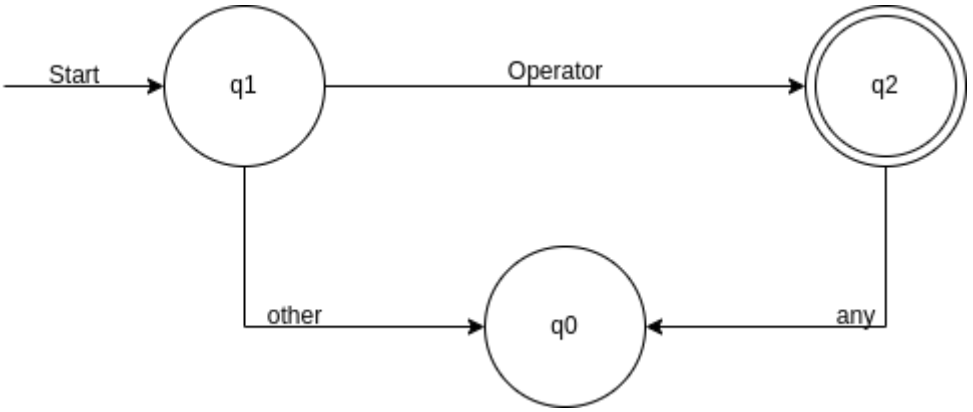
DFA for operators (+, *, =, >):

	Operator	other
Reject (q0)	q0	q0
Start (q1)	q2	q0
q2*	q0	q0

Operator maps {*,+,=,>} and 'other' maps every item of our alphabet differently from the previous column.

* It accepts just one operator and rejects the rest. If we find a "=<", we identify it as 2 tokens and let the parser do what it has to do.

Where the, q1, q2, q3 and q4 states represent the accepting states for the +, *, =, < symbols, respectively, and q0 is the initial state. The q5 state is the accepting state, and indicates that the DFA identifies successfully an operator as valid, while the states marked as "-" indicate that the operator is not accepted as valid by the DFA.



DFA for literals (" "):

q0 represents the starting state, which changes to q1 with an input of " (any other input leads to the rejecting state) and stays in q1 until an input " is received, then changes to q2, the accepting state. Any additional input leads to the rejecting state, as the literal finishes with the second ".

	"	other
Reject (q0)	q0	q0

Start (q1)	q2	q0
q2	q3	q2
q3*	q0	q0

‘other’ maps every item of our alphabet differently from the previous column.