**DEVELOPERS MANUAL**

**Compilers**

**INDEX**

[**1. INTRODUCTION 2**](#_t1ea7okifjed)

[**2. DESIGN AND EXPLANATION 2**](#_99aewkn0qp9x)

[2.1. Structure of the code 2](#_s06e5ulsickv)

[2.1.1. The structure idea 2](#_n57u9cndemf4)

[2.1.2. Pipeline diagram 3](#_qn2wd7kvm5k)

[2.2. The automaton 4](#_2ton7s5kwcue)

[2.2.1. The design 4](#_defal896esnj)

[2.2.2. The transition table 6](#_t9a3hkgmzcs3)

[2.3. Number of operations 8](#_pejt1d8253gt)

[2.4. Error Handling 8](#_2vxqnec3rf7p)

[**3. CODE AND IMPLEMENTATION 9**](#_vc43ud8ds593)

[3.1. The automaton 9](#_5a1uxkixdif1)

[3.2. Read in the input file 9](#_8ha8ffc8tg5y)

[3.3 Token List (in memory) 9](#_vn1xqax5p7fq)

[3.3.1. New line token 9](#_26mhhzxf646a)

[3.5. Debug option of output 10](#_v30ughhbgfp6)

[3.6. Error handling 10](#_fuyd7as7zxpt)

[3.7. Number of operations 11](#_juxn66ebn253)

[3.8. Implementation 11](#_bpxngk3qdztq)

[3.8.1. Parser 11](#_9qn0u9hvypej)

[3.8.2. File distribution and libraries 11](#_3we40giqxphu)

[3.9. Testing 12](#_oozjsuvzxnv3)

[3.9.1. File test.c 12](#_fv3qd4fgh90o)

[3.9.2. File test2.c 12](#_7f5pbimmxotb)

# **INTRODUCTION**

In this practice we work the lexical analysis by implementing a simple scanner.

Lexical analysis involves breaking down the source code into a sequence of tokens for further processing. Our focus will be on crafting a simple yet effective scanner, which plays an important role in recognizing and categorizing these lexical units.

# **DESIGN AND EXPLANATION**

## **2.1. Structure of the code**

## **2.1.1. The structure idea**

The main code will load a file, and will get its arguments.

The organization of the functions and the connection between them is as follows:

* The tokenize function will receive as input parameter a FILE pointer f, that will be the file we will be processing and tokenizing.

It will create a list calling the automata function.

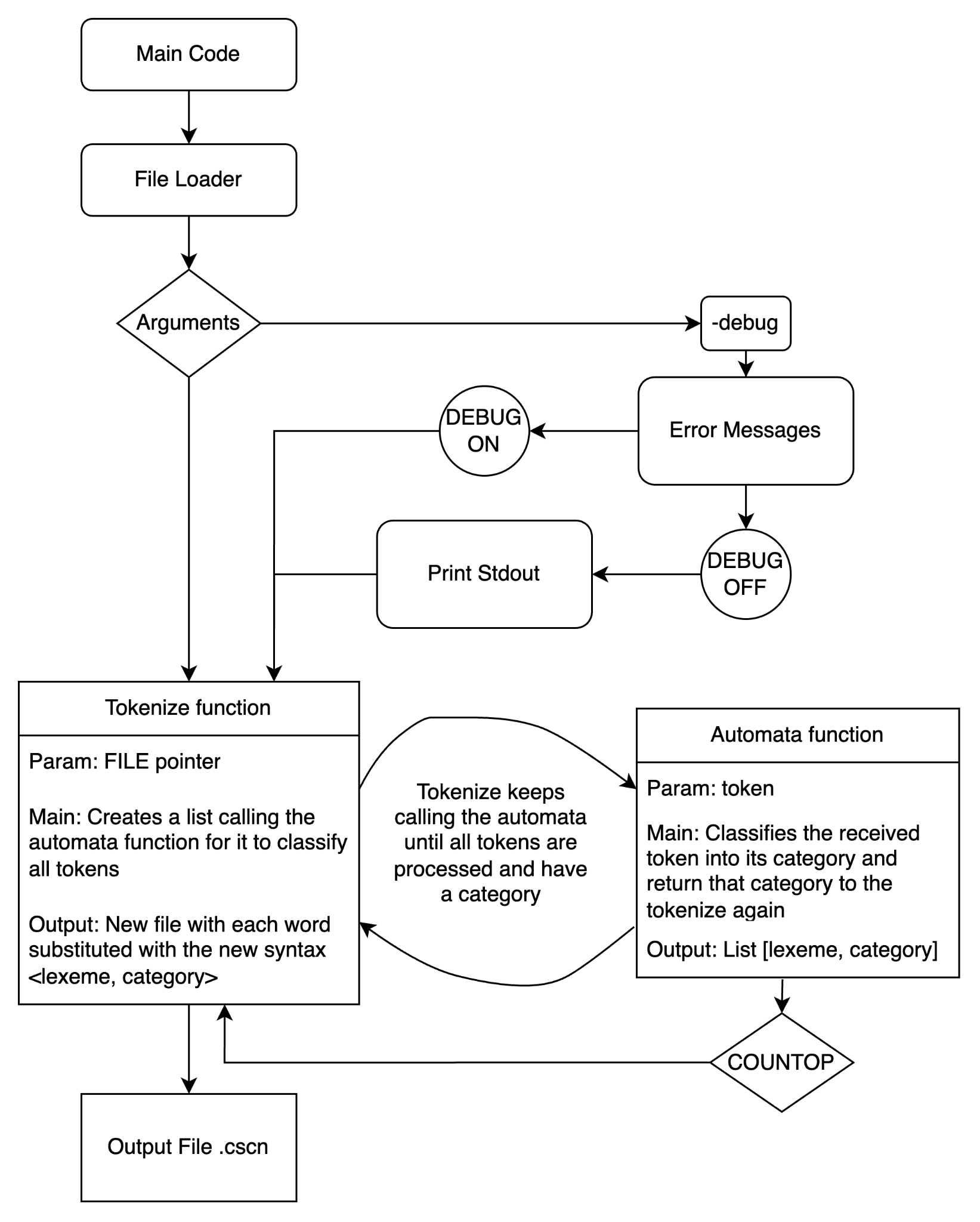
* The automata function will receive the token that the tokenize function passed to it.

It will process that token char by char and, depending on which accepting state it gets (if it gets to any accepting state), it will assign that token a category.

After assigning a category to the token, it will return back again to the tokenize function the output in the corresponding format: [lexeme, category].

* We will repeat this process once and again until all tokens from FILE f are processed by the automata and returned in the correct syntax to the tokenize.
* We will keep track of the number of operations we are performing inside the automata with a counter COUNTOP, just as an indicator for ourselves to be aware of the amount of operations we are computing, to see if we can do it in a more efficient way.
* The list of all the outputs returned to the tokenize by the automata is what will be copied in the output file .csn.

### **2.1.2. Pipeline diagram**



## **2.2. The automaton**

### **2.2.1. The design**

For the automaton we will use an alphabet denoted by the union of the next subsets:

When designing the functionality of the automaton we will search to scan every string that starts with any letter from the alphabet.

If the automaton receives another character from the or subsets it will return the same operator it received as another token with its corresponding value.

If the automaton gets any starting quotation marks it will read the following characters as a string until an end of quotation marks is reached, and pass that string (including the quotations) as the value from the token.

If the string starts with a number and continues with more numbers the automaton will recognise a token with the category of number.

So, concretely, the automaton works as follows (shown by parts for the sake of visualization):

* The automaton will categorize every token from these starting characters. In the following we will divide the number of states shown in the diagrams.





* There will be paths from the characters c, i and v that will drive the automaton to try to identify those strings as indicators of the strings int, char and void. If the transitions are followed we get these strings and then they reach an accepting state after all chars have been read.
* The same will be applied to the other characters e, m, r, w and finally an additional path on i to identify strings like else, main, if, return and while.



* Finally, if they were followed by any character from the or subset, excluding those characters after the minus sign, the automaton will find those strings as identifier words. The start of a word with any character (excepting the ones mentioned) is also considered.



* To the other remaining characters, their representation is the following:

In the case of the set λ we exclude from all the writable characters the quotation marks.



So, as it is said at the beginning, this automata is shown by parts but the implementation is only one big automata that can be seen by the names of the states, having a final automata of 33 states, from 0 to 32, where the last state is included to reach a final non-accepting state and ensure completeness. This will lead to the category CAT\_NONRECOGNIZED.

We need to create constants for each token category:

* CAT\_NUMBER
* CAT\_IDENTIFIER
* CAT\_KEYWORD
* CAT\_LITERAL
* CAT\_OPERAND
* CAT\_SPECIALCHAR
* CAT\_NONRECOGNIZED

### **2.2.2. The transition table**

After joining all our previously shown pieces of the automaton into a big single DFA, we get the following transition table that we will follow in our code implementation. The automaton could have been splitted into a set of smaller automatas however in order to have a single transition table to query we made a big one.

To understand the matrix that we designed we have decomposed it into chunks of smaller size. The columns represent the different characters (our code will translate the ASCII value of each character into a column position. The rows are the states, where the initial state is . For example if we are in the initial state and we read the charchar A we will translate the ASCII value of A which is 65 into a column position, namely 17, and see the corresponding transition in that position 0x17.

By doing so we obtained a matrix with a total of 74 columns ordered like follows.

The transformation between the ASCII value and the column position is done with the function charCol which takes a char and returns the value of the column of the designed matrix.

|  | **(** | **)** | **\*** | **+** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **;** | **=** | **>** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ASCII value** | 40 | 41 | 42 | 43 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 59 | 61 | 62 |
| **column position** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

|  | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ASCII value** | 65 | 66 | 67 | 68 | 60 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 | 81 |
| **column position** | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 |

|  | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z** | **[** | **]** | **a** | **b** | **c** | **d** | **e** | **f** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ASCII value** | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 93 | 97 | 98 | 99 | 100 | 101 | 102 |
| **column position** | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |

|  | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ASCII value** | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 | 112 | 113 | 114 | 115 |
| **column position** | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

|  | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **{** | **}** | **“** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ASCII value** | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 125 | 34 |
| **column position** | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 |

Notice that this simply represents the notation of the columns and the rows, whilst the matrix is generated with a function called initStates with which we initialize the states of the matrix as we described in the diagram.

For example, in the diagram we can see that when in it reads the char ‘r’ it goes into state . In consequence, when we look at row 0 and column ‘r’→ 114 in ASCII → column number 62 we will find the value 1 which means that we go to state 1.

To check if the ending state is an accepting or rejecting state we’ve got the function called checkState which simply checks if a given state is an accepting state or not by comparing it with those defined as accepting states in the diagram.

The function advState will return the next state by querying the design matrix and return the next state. This process is as described in the two paragraphs above.

## **2.3. Number of operations**

To count the number of operations we used a global variable countop initialized to zero. This variable is incremented by 1 for every operation made by the automata, printed out and reset to zero for the next set of operations the automata will do.

## **2.4. Error Handling**

We will handle errors depending on the type of error, and return an error type with the step in which the error occurred.

Firstly we will handle the input file f, ensuring that anything other than the .c file will not be executed.

Then we will check if this .c file is correct, has text and can be opened without trouble. If it has text, the next error we will handle is the one that can happen when parsing and creating tokens.

Finally any error that happens when writing the output file and creating the tokens list will be handled too.

# **CODE AND IMPLEMENTATION**

## **3.1. The automaton**

The automaton, as explained later, will start reading the first character of the token.

* If the first character is an operator or an operation sign it will accept that only character.
* If it is a quotation mark it will accept everything written inside the quotations until there is a finishing quotation mark.
* If it is a number will accept that number and any contiguous numbers.

Then if the first character is a letter the automaton will consider the following actions:

* If the letter is r,m,i,c,v,e,w the automaton will first consider if it is because it has found a keyword starting with one of those letters: char, int, else…
* If the next key from the previous stated is not the next to form the keyword (for example if it founds a *c* and it does not follow with *h*)then the automaton will consider that as a variable identification and will accept any character excluding the one to form the keyword at first, (if it starts with *c* the next letter can not be an *h* as stated in the handout) and then accepting all characters possible from numbers or letters (if it finds for example *ca* for example now any letter or number can be added as it can not form the word *char*).
* Finally if the word being read starts with any letter excluding r,m,i,c,v,e,w it will consider the character and any possible letter or number after

## **3.2. Read in the input file**

The file will be opened through a pointer in read mode.

In the main we will process the arguments and we will check that the input file is a “.c” file. As we cannot use the <string.h> library, we will create a function that does the work of comparing 2 strings to avoid using strcmp.

## **3.3 Token List (in memory)**

After the automata assigns a category to the token that the tokenize function has previously passed to it, the automata will return back to the tokenize function the output in the format: [lexeme, category], where lexeme corresponds to the token, and category corresponds to its category among the available ones.

When all the input file f has been processed by the automata, it means we already will have all the tokens together with their category. Moreover, we need to output the list of tokens in the format <lexeme, category> into an output file, that will be the one with .csn.

It is important to write the tokens in the same order and same line as the input file.

### **3.3.1. New line token**

We created a new token called CAT\_NEWLINE that is not printed in the output file. We use this token to print the ‘\n’ inside the file so that we have the same lines in the output file and the input file.

**3.4. Create output file/release**

We need to create the output file that will contain the token list.

To do so, this function takes the token list created before in the tokenize function and writes it to the output file. For that, we will open the output file for writing, it needs to have the ".scn" suffix to the input file name, and then just write the tokens to the file.

However, before writing the tokens to the output file we will remove any empty lines from the input file to ensure that we are writing the tokens in the specified output format, the Release format.

## **3.5. Debug option of output**

We have controlled the debug mode with the DEBUG macro.

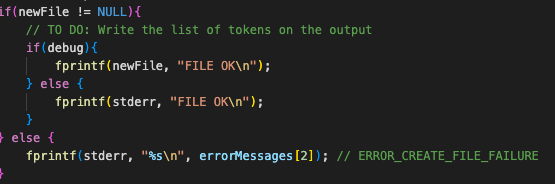
We can change the configuration here:



When the debug mode is on or 1. The variable debug is initialized. So when an error occurs during the file reading or creation process, the program debugs the output and error message to the new file. Additionally, if it is created in the right way, it also outputs “FILE OK\n”.

But when the debug mode is off or 1, the variable debug is not initialized or initialized to false. That is when the error messages are printed to stderr.

For example, when we create a new file, when the file is created successfully and the debug is on, we print FILE OK in the newFile, but when it is off, we print it to the stderr.



## **3.6. Error handling**

We have detected primarily three types of errors.

| Number | Type of Error | Message of Error |
| --- | --- | --- |
| 0 | Parameter | “File parameter failure” |
| 1 | Read | “Read parameter failure” |
| 2 | Create | “Create parameter failure” |

We have initialized an array of pointers to constant characters strings. The const char\* errorMessages[], declares an array of pointers. The first element of the array, saves the character string “File parameter failure”. The second element, assigns “Read file failure”, and finally the third and last element, “Create file failure”.



With this array, in the code, if someone wants to indicate an error happening, instead of writing again. They just need to call the errorMessages with the corresponding position of the error.

## **3.7. Number of operations**

To implement the counting of operations we first are going to create a constant COUNTOP that can be ON (1) or OFF (0). When we do the identifications of tokens, we are going to create a variable that is going to be a counter, this counter is going to increase every time a comparison operation occurs, this should be applied when COUNTOP is ON.

After processing each token, if COUNTOP is ON, we are going to print a message indicating the number of comparison operations performed to identify all tokens.

## **3.8. Implementation**

### **3.8.1. Parser**

To implement a parser we will need to use a recursive descent parsing technique. We will need to:

* Define a structure able to recognize different types of tokens that a parser can recognize.
* Pass an input string.
* Process it and read the input string
* Return the next token encountered.

To get the next token we will create a function able to tokenize the string. It will read characters from the input string and convert them into tokens. Then in the end we will create a function able to evaluate the parser expressions.

### **3.8.2. File distribution and libraries**

The code will be distributed in the next files:

* functions.c
* functions.h
* man.txt
* scanner.c
* scanner.h

Also, the libraries we will use are: <stdio.h>, <stdlib.h>, <ctype.h>

## **3.9. Testing**

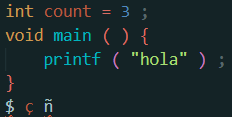
We must clarify that our scanner is working for input files that have the test code separated by spaces.

### **3.9.1. File test.c**

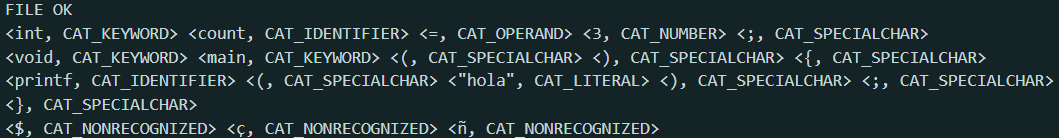
The first test, in the file named *test.c*, is a very simple piece of code to test all possible categories.

We find a keyword *int*, an identifier *count*, an operand *=*, a number *3*, another two keywords *void* and *main*, some special characters such as *;*, *(*, *)*, *{*, *}*, a literal *hola*, and finally non-recognized *$, ç, ñ*, as they are not part of the alphabet.

input:



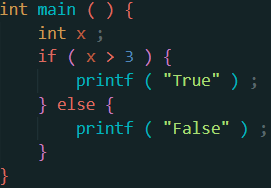
output:

****

### **3.9.2. File test2.c**

The second test, in the file named *test2.c*, has been extracted from one of the examples given in the handout, in page 3. The reason for doing this test is that we can make sure if our code is doing exactly what it is expected to do, as we have the output we should get.

input:



output: