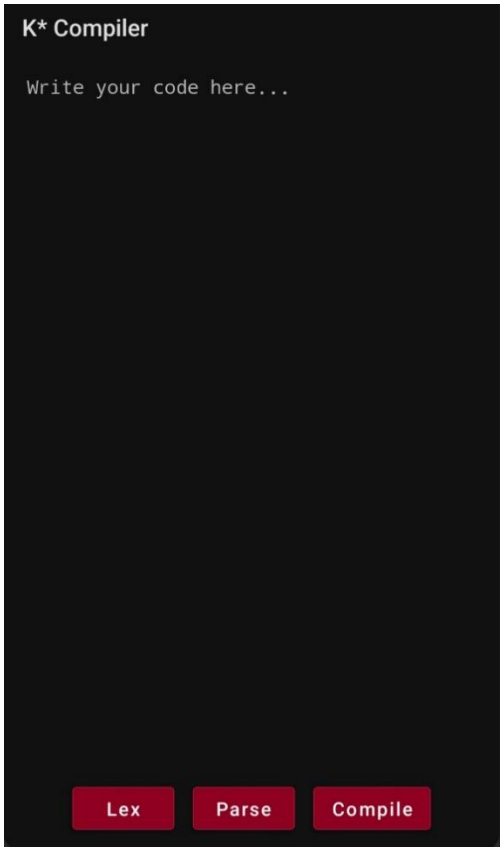
- **Main interface – Installation and Use.**

When downloading the application, it will appear on our device showing the following icon and name in your phone's menu.

When entering the application, the following main interface will be displayed, consisting of an environment in which the code to be processed will be entered, and the three main functions that we developed and tested for our compiler.



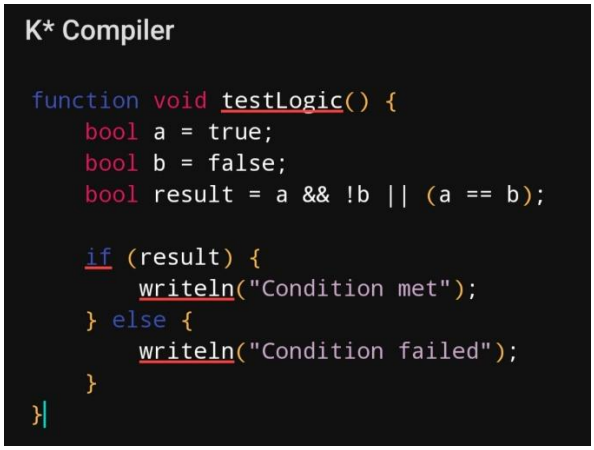
- **Lex button (Lexical Analysis)**

When the Lex option is selected, the compiler performs a lexical analysis to the source code we entered. At this stage, the program is read character by character to identify the lexemes. Each lexeme is classified into a token, such as identifiers, keywords, operators, literals, and others.

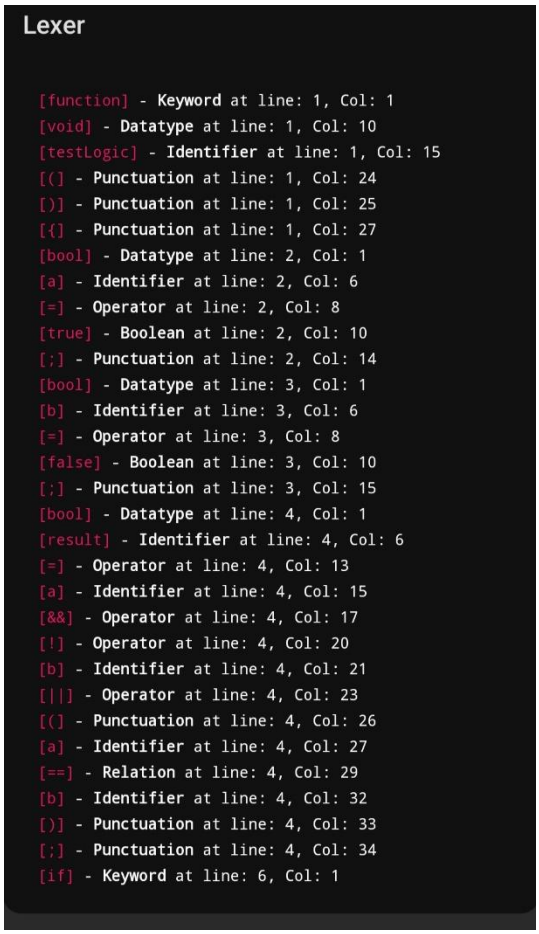
This phase allows us to verify if the code contains valid tokens according to the defined language. The result is a stream of tokens that will later be used in the parsing stage.

When entering the code into the application, it is important to make sure that the syntax defined by the KStar language is respected. This includes the correct use of keywords, delimiters such as braces and semicolons, as well as the proper order of statements. In addition, the use of characters not recognized by the lexer (e.g., special symbols or improperly closed quotation marks) should be avoided, as they could generate errors during lexical or syntactic analysis.

Example: We enter the following code fragment, which illustrates the use of multiple token types in the context of a function structure, a conditional structure and function calls.



The program shows the following output, which contains each of the lexemes classified based on the available tokens, as well as their location within the input code, specifying their line and column in which they are found.



It is important to make clear that, although the code entered is processed correctly with this first compiler option, it is not guaranteed that the following two options can correctly process this same code fragment. Each fase will have its own requirements to take in consideration if we want the program to work properly.

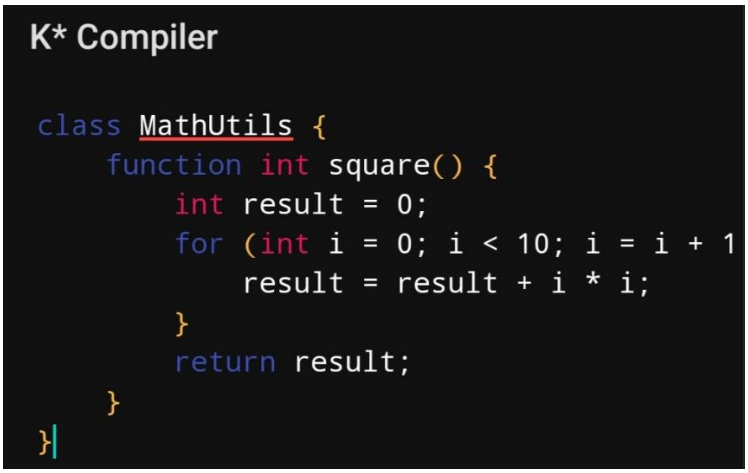
- **Parse Button (Syntactical Analysis)**

The “Parse” option takes as input the token stream generated by the lexical analysis and compares it against the specified language grammar. At this stage, the compiler validates whether the structure of the source code follows the predefined syntax rules, such as the correct order of statements, expressions, blocks, functions, classes, etc.

If the program syntax is valid, the parser builds a structured representation of the code, such as a parse tree, which will serve as the staple for the following stages of the compilation process. If there are errors, messages are displayed indicating in which part of the code they were found.

When we are using the parser, again, it is essential to know that the source code is structured according to the grammatical rules of the KStar language. This means that constructs must follow the hierarchical order defined in the grammar, such as placing statements inside classes or functions, and ensuring that expressions are correctly nested. Any omission of mandatory elements, such as parentheses in conditions or semicolons at the end of expressions, can cause errors in parsing. Also, the parser assumes that the lexical analysis was successful, so the code must have been previously validated with the Lex module to avoid errors.

Example: We will enter the following code fragment, which defines a class called MathUtils that contains a function called square. This function declares a variable result that is initialized to zero, then uses a for loop to iterate through the numbers 0 through 9.



After processing this code with the parse option, the parser builds the parse tree reflecting the program hierarchy by means of derivation steps. The root node will correspond to the MathUtils class declaration, which will contain a child node representing the square function. This in turn will include child nodes for the variable declaration, the for loop, the update expression, the loop body, and finally the return. The output must confirm that the code structure is syntactically valid and that it respects the rules defined by the language grammar.

At the beginning of the output, a message will be displayed confirming that the parsing process was completed successfully.

At the end of the output generated, a message confirming the absence of semantic errors in our code will be displayed.

- **Compile Button (Code Generation and Execution)**

The Compile button represents the final stage of code processing in the application. When selected, the compiler takes as input the code previously analyzed lexically and syntactically and the application proceeds to generate its simulated execution. This phase involves the semantic validation and logical execution of the program, allowing us to observe the real behavior of the program according to the rules of the defined language.

It is important to clarify that, for the correct use of this option in our program, it is necessary to write a code with instructions to be executed already defined, together with the operations that we wish to perform with it. For the previous phases, it was only possible to define classes, functions, and other structures to be executed without really receiving an indication or parameters to work with, which if done with this last option, will result in an error thrown by the program since it directly executes the entered code waiting for parameters ready to throw a result already processed, which we will see with the next example.

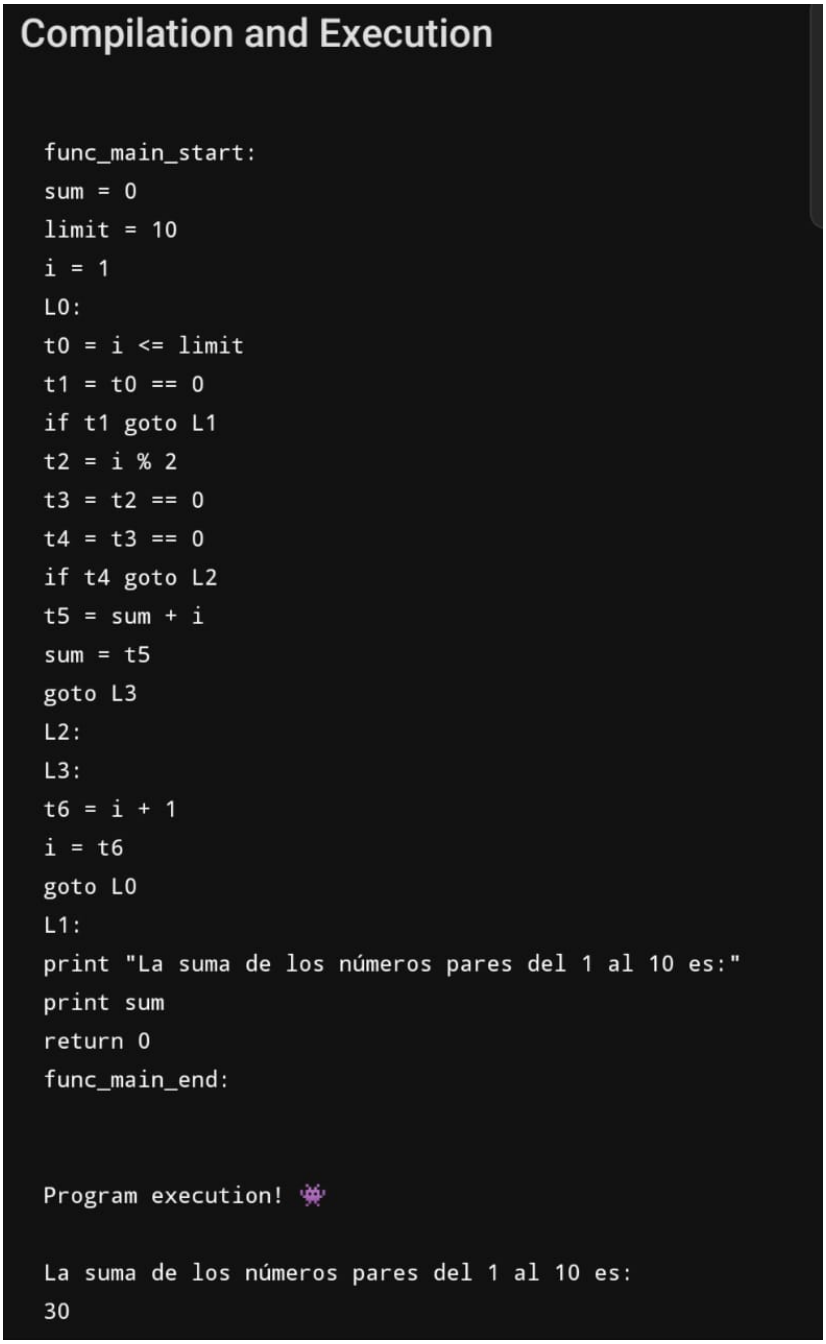
We write the following program, which declares two variables: sum to store the sum of even numbers, and limit with the value of ten. Then, by means of a for loop, it runs through the numbers from one to ten. At the end, it prints the result on the screen.

K* Compiler

```
function int main() {
    int sum = 0;
    int limit = 10;

    for (int i = 1; i <= limit; i = i + 1) {
        if (i % 2 == 0) {
            sum = sum + i;
        }
    }

    writeln("La suma de los números pares del 1 al 10 es:");
    writeln(sum);

    return 0;
}
```

We can see that this is a program with an already defined flow of execution that makes use of the declared structures.

By pressing the third button, we get the following output:



When the program is compiled, the application generates an intermediate representation of the program flow in the form of three-address code instructions, followed by their simulated execution. In this case, the code calculates the sum of odd numbers from one to ten using a for control structure and nested conditionals. The output first shows the sequence of instructions generated, including assignments, arithmetic operations, logical evaluations and conditional jumps, which we can see labeled as L0, L1, etc., which facilitates the analysis of the program flow. Finally, the result of the execution is presented, which prints on the console

the message: “The sum of the even numbers from 1 to 10 is:”, followed by the calculated value: thirty. This shows that the compiler not only analyzes and translates the source code correctly, but also simulates its behavior.