Programming Course Project
ITALO CODE

Diana Rubaga
Gregorio Orlando
Alberto Puliga

On 17 October 2023

BCSAI

Table of Contents

1. Project Planning:

Aim:
C programming is known to be a complex language to understand and use, and has been a building block for many other languages such as Python etc. This project is focusing on the use and the extension of C programming, to create a language that is simple and user-friendly and is based on the basic structure and foundation of C. This will allow us to better understand the inner mechanism of what makes this language and how we can use this to then create a simpler language with italian tokens. Additionally, this project will widen our knowledge regarding lexer, interpreters, and parsers which can be super useful in the future.

Main Features of the Language:
The language we have chosen is an Italian programming language with a clear relationship to mathematics for beginners. This relationship was established because it not only allows the Italian user to have access to a low code language but also the universal language of mathematics to perhaps other users/readers to also understand and use this language. The idea of this project derived from the fact that we are a group of Italians, that, as students of Computer Science, understand how at the beginning of someone's coding journey it can be a hard and terrifying hurdle. Especially when most languages have English keywords making it hard for foreign speakers to understand the sheer logic of the code because rather they focus on the translation. Therefore, we are creating a language that is suitable for Italian coders that are beginners, or just people who enjoy mathematics and want to learn.

- **Purpose**: The main purpose of this project is to bridge the gap between Italian speakers, mathematics, and programming, thus sending a message that coding is accessible to everyone. Additionally, another main idea is that this could be used in the Italian education system, because as of now the Italian education system, that we attend, did not understand the importance of teaching technological evolution. Thus with this programming language perhaps it can become more integrated, allowing students to explore at a young age what coding is.

- **Simplicity**: This language is not supposed to be used to create complex algorithms, the simplicity is what is important. We will enforce this simplicity by using familiar phrases and easy-to-understand syntax that will make coding seem like spontaneous talking and writing, like how it's super easy to understand basic mathematical concepts.
- **Scalability**: a basic set of tokens (keywords) are currently functional in our code; on top of that many tokens have been implemented already in the lexer to allow for more advanced features such as boolean operators and even more can easily be added. This is done to make sure the difficulty level of our programming language continually matches the skill and capability of the user and is able to grow accordingly.
- **Target Audience:** The main target audience are Italian speakers/natives, perhaps Italian education institutions wanting to expand their students' knowledge to better suit the evolving worlds, and lastly anyone who likes mathematics and wants a creative way to also learn coding, young Italian students that want to learn the logic behind coding. The main idea is that these languages target beginner coders, with a background of Italian languages or mathematics.

2. Language Design:

For this Programming Language project, referencing back to the target audience and the simplicity needed in the connection of Italian cooking it is important to have clear syntax.

CIAO()	CIAOCIAO()	PS	E	O
Start	End	Comments	and	or
DATA types = primary food types in italian food				KeyWord
CARBS	VEGGIE	FORMAGGIO	OLIO	Frutta (keyword)
int	float	char	string	void z
KEYWORD	ARITHMETIC OPERATIONS = silverware in italian			
Scrivi	piu	meno	per	diviso
printf()	+	-	*	//

End Of Line character (EOL): !

General brace operators : ( )

3. Create a Project Plan:

To have a successful project we must create a project plan to understand the project better and meet these requirements and also have good time management skills. The table below highlights the tasks rated on importance, and manageable steps that will have to be taken to complete this task. This will help keep the group accountable in completing their tasks and not to be overwhelmed.

Task name:	STEPS:	Importance level:	Due date:	Name
Language Design:	Make design setup and syntax	MEDIUM	17 November 2023	Diana
General document and Documentation:	Brainstorm as a group	MEDIUM	17 November 2023	Diana
Build Lexer (Tokenizer):	Steps: Define Token Types Create Lexer Rules	HIGH	20 November	Diana

Task name:	STEPS:	Importance level:	Due date:	Name
Start by understanding the concept of a lexers and tokenizer	Initialize Lexer State Lexical Analysis Loop		2023	
Build a Parser: Start by understanding the concept of a parser and how it connect to lexer	Steps: Grammar of the Language Choose the Type of Parse Handle Tokens from the Lexer	HIGH	21 November 2023	Albi
Implement the Interpreter: Start by understanding the concept of a interpreter and how it connect to the previous implementations	Steps: Define the Language Specification Design the Architecture Implement a Lexer Implement a Parser Build or Generate an Abstract Syntax Tree (AST)	HIGH	23 November 2023	Gregorio
Error handling:	Understand error handling in c programing	MEDIUM	25 November 2023	Gregorio
Sample code creation	Run the code and smooth the process	MEDIUM	30 November 2023	Diana+Gregorio +Abli

4. Developing a Lexer:

A lexical analyzer known as lexer or tokenizer is a component in the processing languages that helps implement tokenizing towards the input code into meaningful components.

- Write regular expressions or a similar method to recognize keywords, identifiers, literals, and symbols.

Unique keywords like "PIU", "MENO", "STAMPA", etc... indicate that the lexer is customised to a subset of token types, probably for a language specific to a certain domain. This illustrates how a lexer works by dividing an input string into identifiable tokens that a parser in a compiler or interpreter can subsequently process.

Output and testing of lexer:

```c
int test() {
    const char *input = "12 piu oppure ciaociao ( )";
    lexer(input);
    return 0;
}       You, 2 days ago
void main() {test();}
```

```
PROBLEMS 5   TERMINAL   OUTPUT   DEBUG CONSOLE   PORTS   GITLENS   AZURE   COMMENTS

cd "/Users/albertopuliga/Desktop/Coding/GitHub/ItaloCode/src/" && gcc lexer.c -o lexer && "/Users/albertopuliga/Desktop/Coding/GitHub/ItaloCode/src/"lexer
 ~/Desktop/Coding/GitHub/ItaloCode  main !1
> cd "/Users/albertopuliga/Desktop/Coding/GitHub/ItaloCode/src/" && gcc lexer.c -o lexer && "/Users/albertopuliga/Desktop/Coding/GitHub/ItaloCode/src/"lexer
lexer.c:116:1: warning: return type of 'main' is not 'int' [-Wmain-return-type]
void main() {test();}
^
lexer.c:116:1: note: change return type to 'int'
void main() {test();}
^~~~
int
1 warning generated.
 Lexeme: 12, TokenName: NUMERO, Token: 8
 Lexeme: piu, TokenName: PIU, Token: 12
 Lexeme: oppure, TokenName: OPPURE, Token: 11
 Lexeme: ciaociao, TokenName: CIAOCIAO, Token: 1
 Lexeme: (, TokenName: OPEN_PARENTHESIS, Token: 4
 Lexeme: ), TokenName: CLOSE_PARENTHESIS, Token: 5
```

5. Build a Parser:

Composing an abstract syntax tree (AST) from tokenized input with a parser demands a thorough knowledge of grammar and parsing methods.

- On the other hand, parsing algorithms are not limited to programming languages; they are specifically designed for language grammars as well. A grammar is a system of rules that specify how characters can be put together in a certain order to make sentences that are considered correct within that grammar.

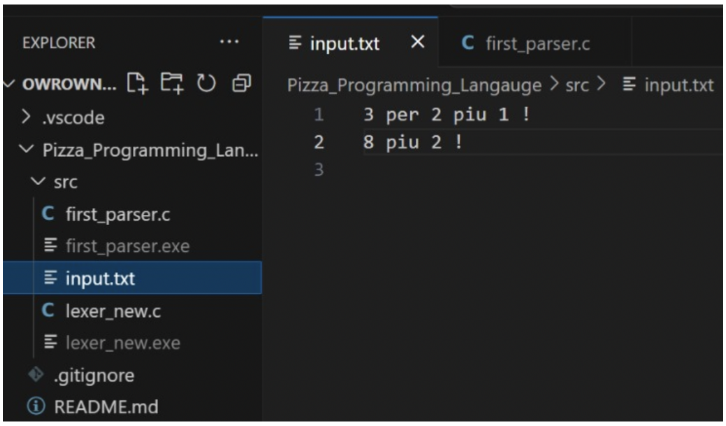
The parser analyses a series of tokens to determine whether or not they follow the rules of the grammar under consideration by utilising a selected parsing technique, such as recursive descent parsing. These tokens are going to be important to be read by our lexer that was explained above. We decided to focus on the top-down parser, which will build the AST by starting its analysis at the top (also known as the start symbol) and working its way down.

Selecting whether to use backtracking is also critical, even though there are other strategies such as leftmost parsing, which gives priority to the expansion of the leftmost non-terminal node.

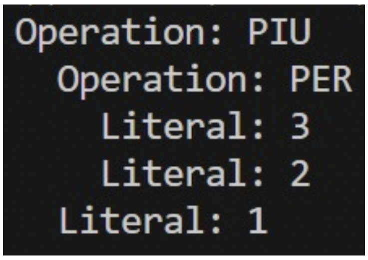
Even though backtracking is a flexible programming technique, it can increase complexity, so its necessity and effect on the parsing process should be carefully considered. As a result, creating this parser requires not only technical expertise but also the ability to make strategic decisions about parsing techniques and the nuances of the particular grammar under consideration.

Below you can see with our sample input of adding and subtracting using our tokens from our lexer and how our input will print out the corresponding tree with the correct priorities. This is just showing that our parser works however just for these token as a more in depth testing will be deployed later on in the document:

Input in our language:



Output the printed parsing tree:



6. Build a Interpreter:

The evaluate function in the parser.c file plays a crucial role in evaluating an Abstract Syntax Tree (AST) and returning the result as a string. Because of that, our interpreter is this function, ‘evaluate’. It takes a TreeNode pointer as input, representing a node in the AST, and recursively evaluates the expression represented by the AST. This function is a ‘char’ type, it means it returns strings, this allows us to return the value to print after the token STAMPA or messages for the user, facilitating error handling.

Here's a step-by-step breakdown of how the evaluate function works:

The function checks if the node is NULL. If it is, it means that the AST is empty, and the function returns an empty string by dynamically allocating memory for a single character and setting it to null-terminator ('\0').

The function then enters a switch statement based on the type of the node.

If the node is a LITERAL_NODE or a VARIABLE_NODE, it dynamically allocates memory for the result string, copies the lexeme (value) of the token associated with the node into the result string, and returns it.

If the node is an OPERATION_NODE, it recursively evaluates the left and right child nodes by calling the evaluate function on them. It then checks if either leftValueStr or

rightValueStr is NULL, indicating a memory allocation failure during subexpression evaluation. In such a case, it dynamically allocates memory for an error result string, frees the memory for the partial results (leftValueStr and rightValueStr), and returns the error result string.
If neither leftValueStr nor rightValueStr is NULL, the function converts the strings to integers using the *atoi* function. So it can perform the operations. Once the result is computed it converts the integer to string again so we can return it according to the type of the function 'char'.
Depending on the type of operation (PIU, MENO, PER, or DIVISO), the function performs the corresponding arithmetic operation on the leftValue and rightValue integers. It then dynamically allocates memory for the result string and converts the result value back to a string using the *sprintf* function.
After obtaining the result string, the function frees the memory for the partial results (leftValueStr and rightValueStr) and returns the result string.

If the node is a PRINT_NODE, the function recursively calls the evaluate function on the left child node and returns the result.

If none of the above cases match, the function returns NULL. In fact the return type of the function is a sting and it contains the result of the imputed line.

Overall, the evaluate function is responsible for traversing the AST, evaluating the expressions, and returning the result as a string. It handles memory allocation failures and division by zero errors, ensuring that the program behaves correctly and provides meaningful error messages when necessary.

7. Testing and Sample Program:

The *documentation/video_demo* folder exemplifies the usage of the lexer.c and parser.c, showing how each works behind the scenes.

8. Error handling:

In our programming language, various error handling strategies are employed to ensure the robustness and reliability of the software. These strategies involve the use of conditional statements, dynamic memory allocation, and standard library functions to detect and handle errors. Let's delve deeper into each case:

- Missing Arguments: The code checks for tokens after an operator to detect missing arguments. If a token is absent, it signifies a missing argument for the operation. An error message, which includes the type of the missing argument, is printed to the standard error stream using *fprintf*. The function then returns a message saying where you missed the argument, signalling an error to the calling code. This approach ensures that missing arguments are detected and reported, preventing potential issues or unexpected behaviour. If this error occurs the code continues running and skips the operation of the missing argument.

- NULL Pointer Handling: The code checks if the root pointer is NULL before proceeding with any operations in the *printAST* function. If NULL, the function simply returns, effectively handling the error of passing a NULL pointer. Additionally, the code handles unrecognised node types in the abstract syntax tree (AST) by printing "NODO_NON_RICONOSCIUTO" (UNRECOGNIZED_NODE in Italian) when it encounters an unsupported node type.

- Memory Allocation Failure: In the case of an OPERATION_NODE, the code attempts to evaluate the left and right child nodes. If either evaluation returns NULL, it indicates a memory allocation failure. The code handles this by dynamically allocating memory for an error message using malloc(). The error message "Errore: Memory allocation failure" is then copied into the allocated memory using strcpy(). After this, the memory allocated for the partial results is freed using free(), and the error message is returned. This approach handles memory allocation failures and provides an informative error message to the caller.

- Division by Zero: The code checks if the rightValue is zero before performing a division operation. If it is zero, a division by zero error would occur. To handle this, an error message is printed to the standard error stream using *fprintf*. Additionally, a separate error message is dynamically created using malloc and *strcpy*, and then returned from the function. This allows the caller of the function to handle the error appropriately. The variables leftValueStr and rightValueStr are freed before returning, ensuring proper memory management. A message saying that the division by zero is impossible is displayed and suggests discarding the result.

- File Opening Errors: After attempting to open the file "input.txt" using fopen(), the code checks if the file pointer file is NULL. If it is, the file could not be opened successfully. In this case, the perror() function is called with the error message "Errore nell'apertura del file input.txt". The perror() function prints a descriptive error message to the standard error stream (stderr), along with the string representation of the current error code. This provides a clear error message indicating the failure to open the file, aiding debugging and troubleshooting.

- Line Ending Errors: The code reads and processes each line of a file using a while loop. It checks if each line ends with " !". If a line does not end with " !", an error message is printed to the standard error stream using *fprintf*. The error message indicates that all lines should end with " !" and alerts the user that there is a line without it. After printing the error message, the file is closed using *fclose* and the function returns 1 to indicate an error. This approach allows the program to detect and handle the error condition, providing feedback to the user and preventing further processing of the file.

These detailed error handling strategies ensure that the program can handle a wide range of error conditions and provide informative feedback to the user, enhancing the robustness and reliability of the software.

9. Conclusion and Evaluation:

To run our main file, there is a main function called main, that once ran taking an input.txt where the user has written the code in our language and then runs it through the lexer and parser and interpreter. Which will return the correct output of the mathematical italian equation.

Encouraging Italian speakers and foodies to access and enjoy programming was the driving force behind the design decisions made for your language. The language seeks to reduce entry barriers for novices and offer a culturally relevant learning experience by utilising well-known mathematical terms and Italian language syntax. This method not only shows the potential of fusing cultural components with technical education, but it also innovates in the field of educational programming languages, making learning to code more intuitive for your target audience.

10. Bibliography:

- Typanski, Evan. “Writing a Simple Programming Language from Scratch - Part 1.” DEV Community, DEV Community, dev.to/evantypanski/writing-a-simple-programming-language-from-scratch-part-1-54a2. Accessed 15 Dec. 2023.
- “Recursive Descent Parser - Part 1.” YouTube, uploaded by [channel name], www.youtube.com/watch?v=4m7ubrdbWQU. Accessed 15 Dec. 2023.
- “Recursive Descent Parser - Part 2.” YouTube, uploaded by [channel name], www.youtube.com/watch?v=8Xl4K1JR8kc&list=PL2LwuHMy_JcAKLY_OxrvdY-FPciMPclCs&index=15. Accessed 15 Dec. 2023.
- “Recursive Descent Parser.” Wikipedia, Wikimedia Foundation, en.wikipedia.org/wiki/Recursive_descent_parser. Accessed 15 Dec. 2023.