Kadir Pekdemir 150121069

Burak Karayağlı 150121824

Murat Albayrak 150120025

**CSE2260 - Principles of Programming Languages Project 1 - Part 2**

C programming language was used in this project.

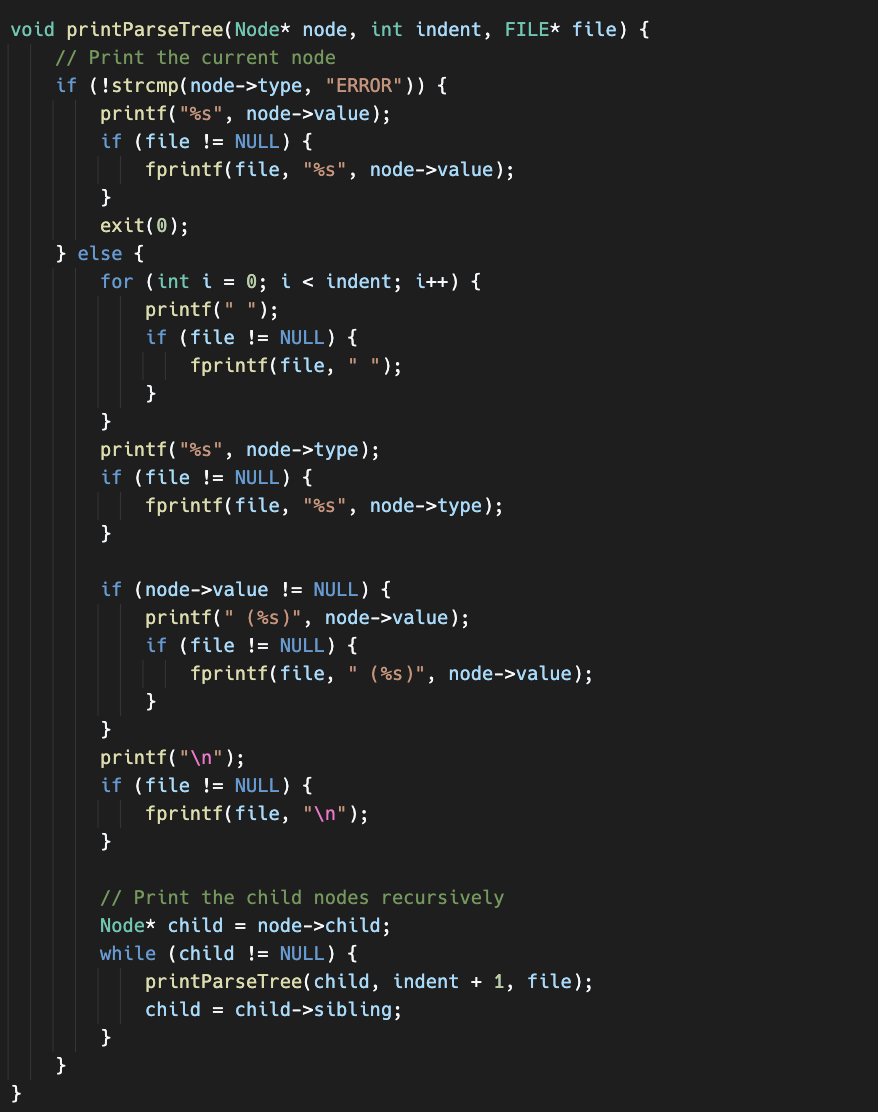
In our project, we will be implementing a syntax analyzer (parser) for the PPLL programming language. The main objective of this project is to develop a recursive-descent parser to analyze the syntax of PPLL programs. By doing so, we can ensure that the programs written in PPLL adhere to the specified grammar rules. This parser will be built upon the tokens produced by the scanner we developed earlier.

There is no part of our code that does not work. We have done all the tasks asked of us.

All functions in the code are explained in detail below.



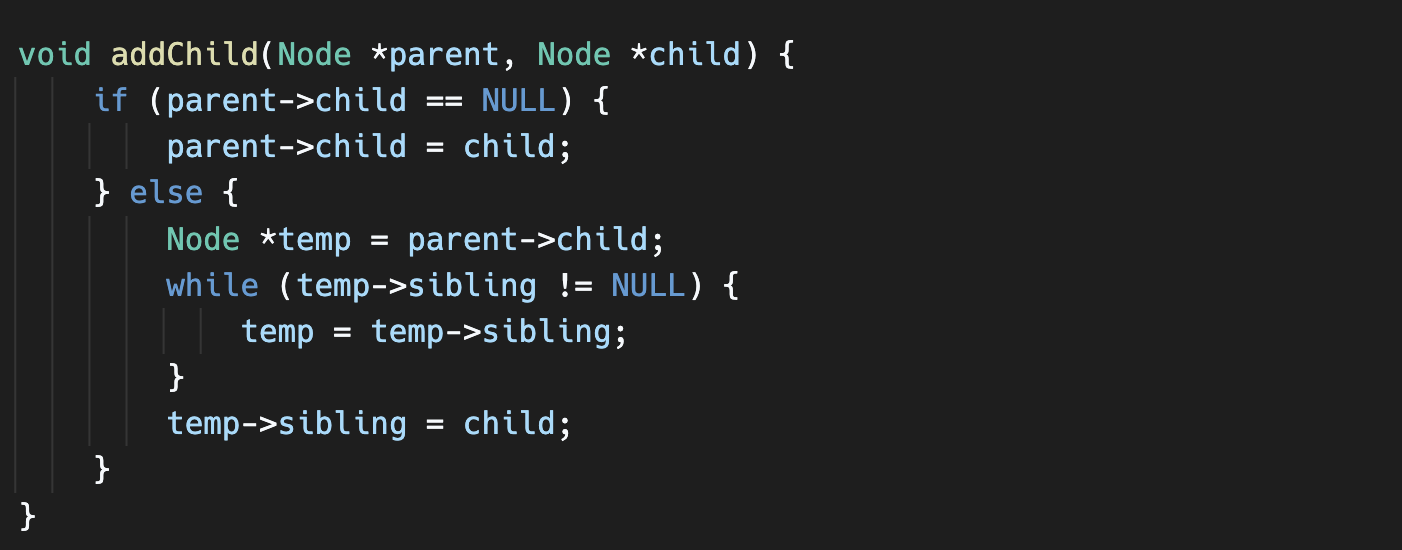
The code defines a struct called **Node** that represents a node in the parse tree. Each **Node** has a **type** and **value** field to store the type and value of the corresponding token. Additionally, each **Node** has a **child** pointer to point to its first child node and a **sibling** pointer to point to its next sibling node.



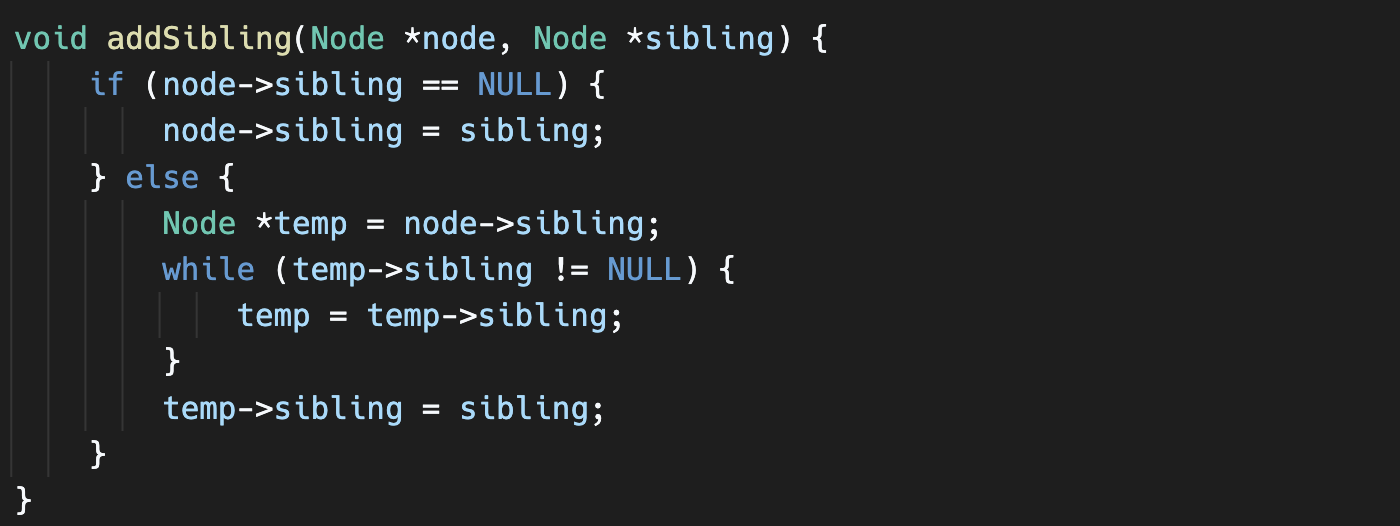
This function is used to properly print nodes representing a tree structure to the screen or to a file. Nodes have a hierarchy and each node can have more than one child node. The function prints the entire tree structure by repeating this process by repeating the child nodes after printing one node.



The **createNode** function creates a new **Node** with the given type and value, and allocates memory for it. The child and sibling pointers of the node are set to **NULL**, and the created node is returned.



The **addChild** function adds a child node to a parent node. If the parent node doesn't have a child, the child node is directly assigned as the child. Otherwise, it traverses the existing siblings of the parent node until it reaches the last sibling, and then adds the new child node as the sibling of the last sibling.



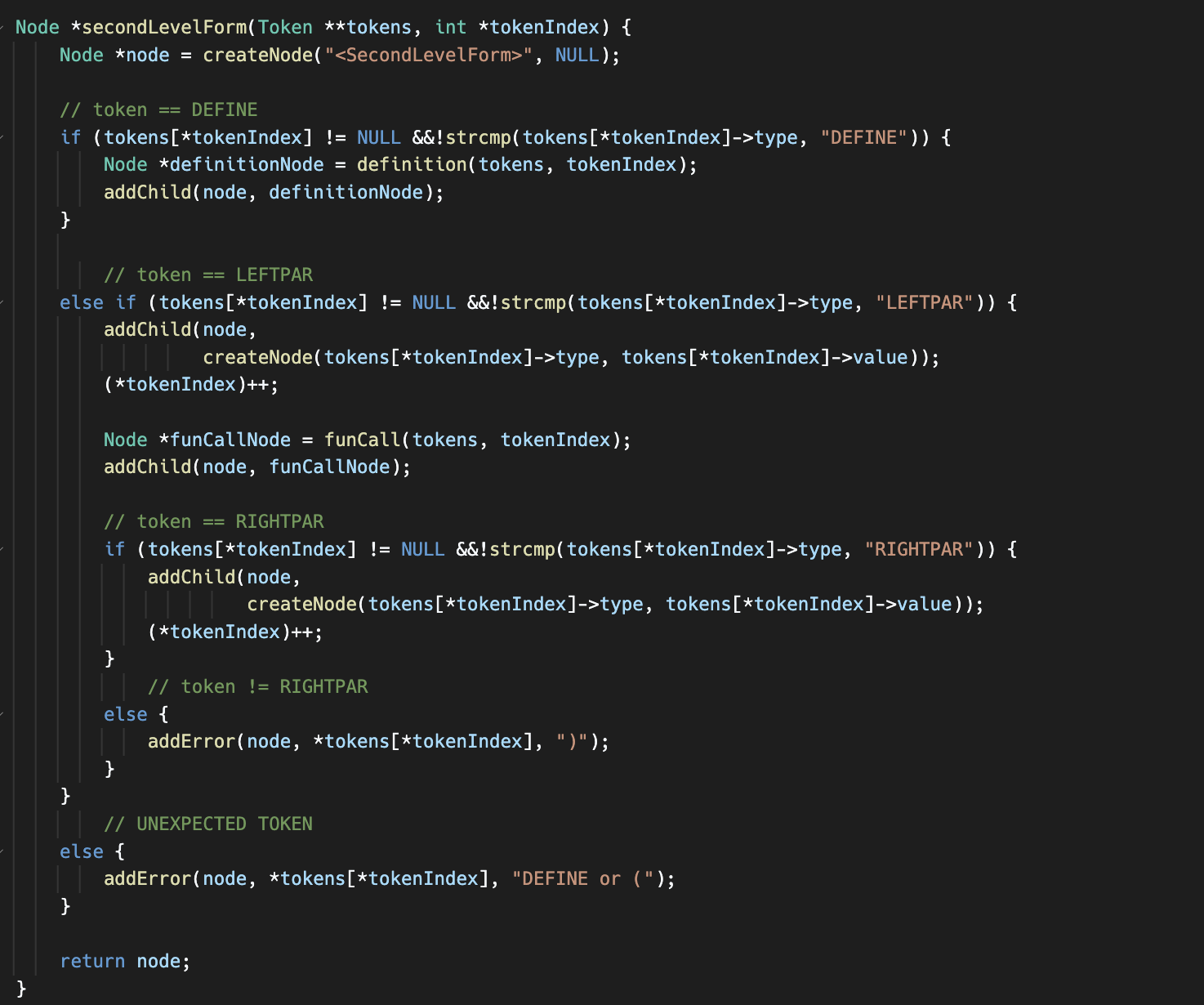
The **addSibling** function adds a sibling node to a node. If the node doesn't have a sibling, the sibling node is directly assigned as the sibling. Otherwise, it traverses the existing siblings of the node until it reaches the last sibling, and then adds the new sibling node as the sibling of the last sibling.



The **program** function takes a **Token** array and a **tokenIndex** counter as parameters. It creates the main program node and then calls the **topLevelForm** function to analyze the top-level forms. The obtained nodes are added to the main program node, and the function recursively calls itself to create the next node. If there is no **LEFTPAR** token at the position indicated by **tokenIndex**, an error node is created.



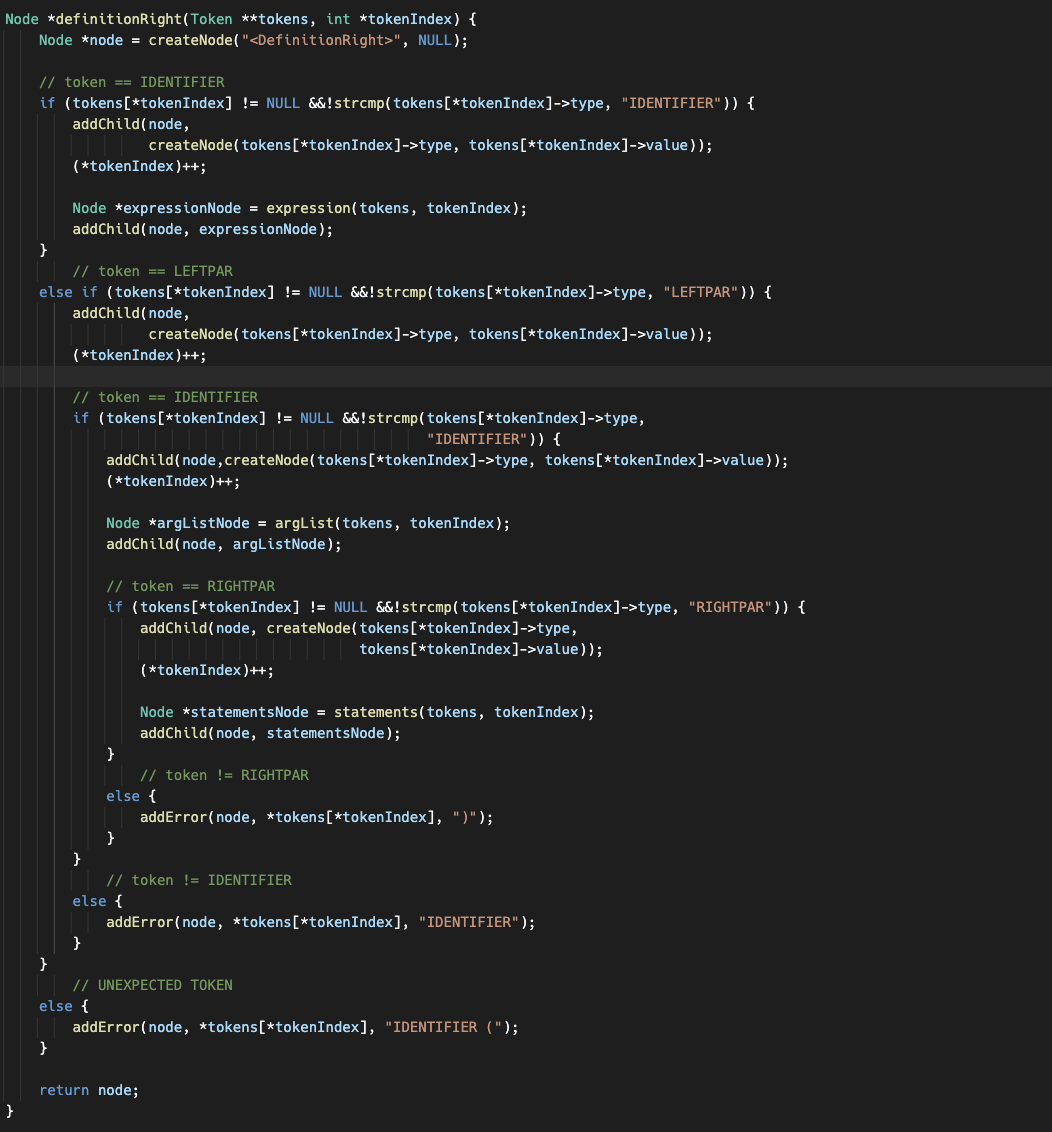
The **topLevelForm** function takes a **Token** array and a **tokenIndex** counter as parameters. It creates nodes to represent top-level forms. It first checks for a **LEFTPAR** token. If the token is present, the corresponding nodes are created, and then it checks for a **RIGHTPAR** token. If the **RIGHTPAR** token is not present, an error node is created.



The **secondLevelForm** function takes a **Token** array and a **tokenIndex** counter as parameters. It creates nodes to represent second-level forms. If a **DEFINE** token is present, it calls the **definition** function and creates the corresponding node. If a **LEFTPAR** token is present, it creates the relevant node and calls the **funCall** function. Then it checks for a **RIGHTPAR** token and handles error cases.



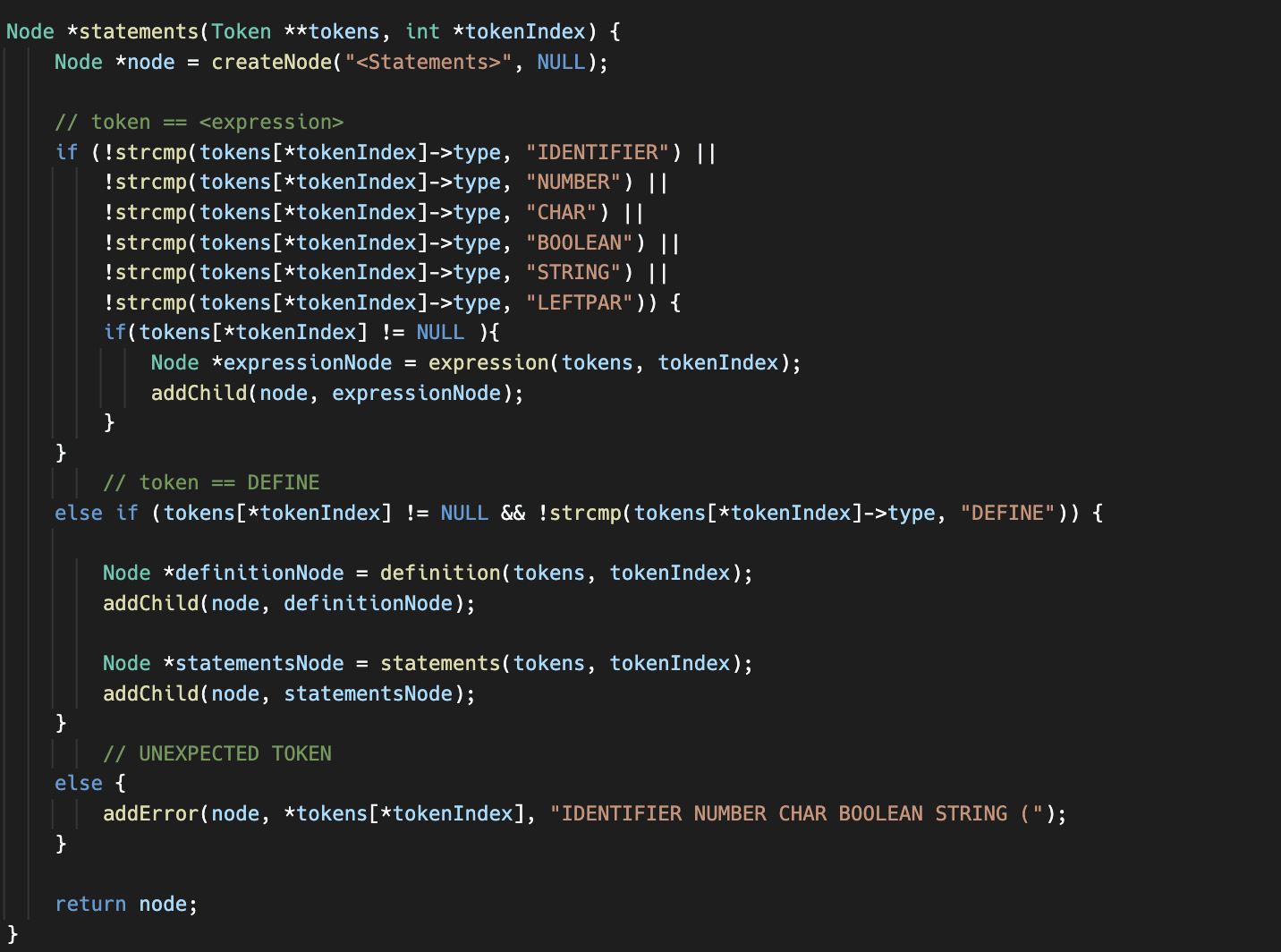
This function creates a node for the **<Definition>** type in the parse tree. It checks if the current token is of type "DEFINE". If it is, it creates a node for the "DEFINE" token and advances the token index. Then, it calls the **definitionRight** function to handle the right side of the definition and adds the resulting node as a child.



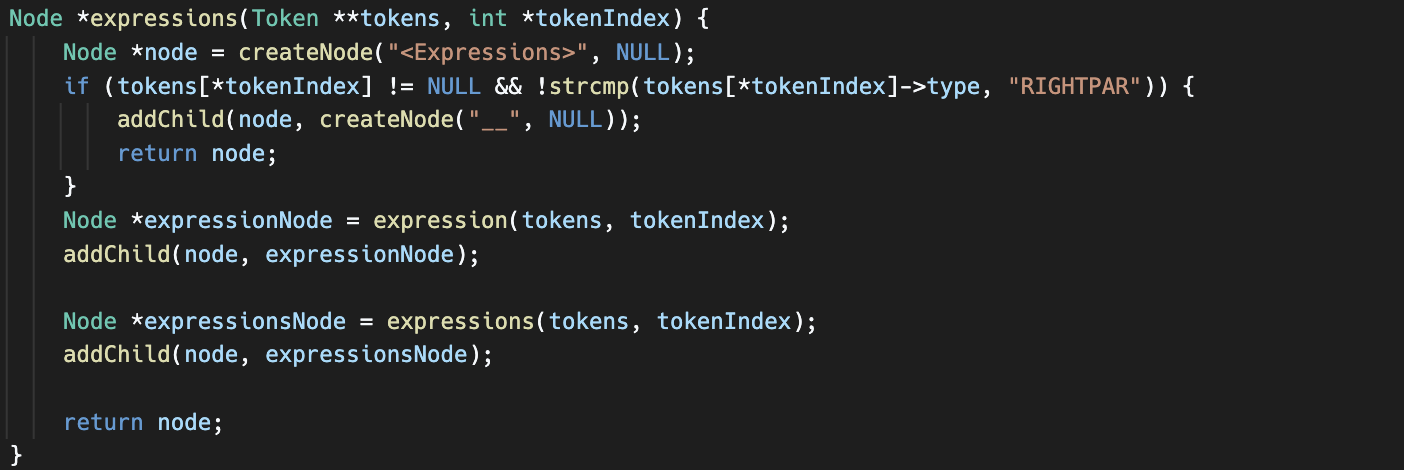
This function creates a node for the **<DefinitionRight>** type in the parse tree. It checks if the current token is either an identifier or a left parenthesis. If it's an identifier, it creates a node for the identifier token and advances the token index. Then, it calls the **expression** function to handle the expression part and adds the resulting node as a child. If the current token is a left parenthesis, it creates the corresponding node, checks if the next token is an identifier, and adds it as a child. It then calls the **argList** function to handle the argument list and adds the resulting node as a child. Finally, it checks for a right parenthesis token and handles error cases accordingly.



This function creates a node for the **<ArgList>** type in the parse tree. It checks if the current token is an identifier. If it is, it creates a node for the identifier token and advances the token index. Then, it recursively calls itself to handle the next argument in the list and adds the resulting node as a child. If the current token is not an identifier, it adds a placeholder node with "\_\_" as the value. The function returns the created node.



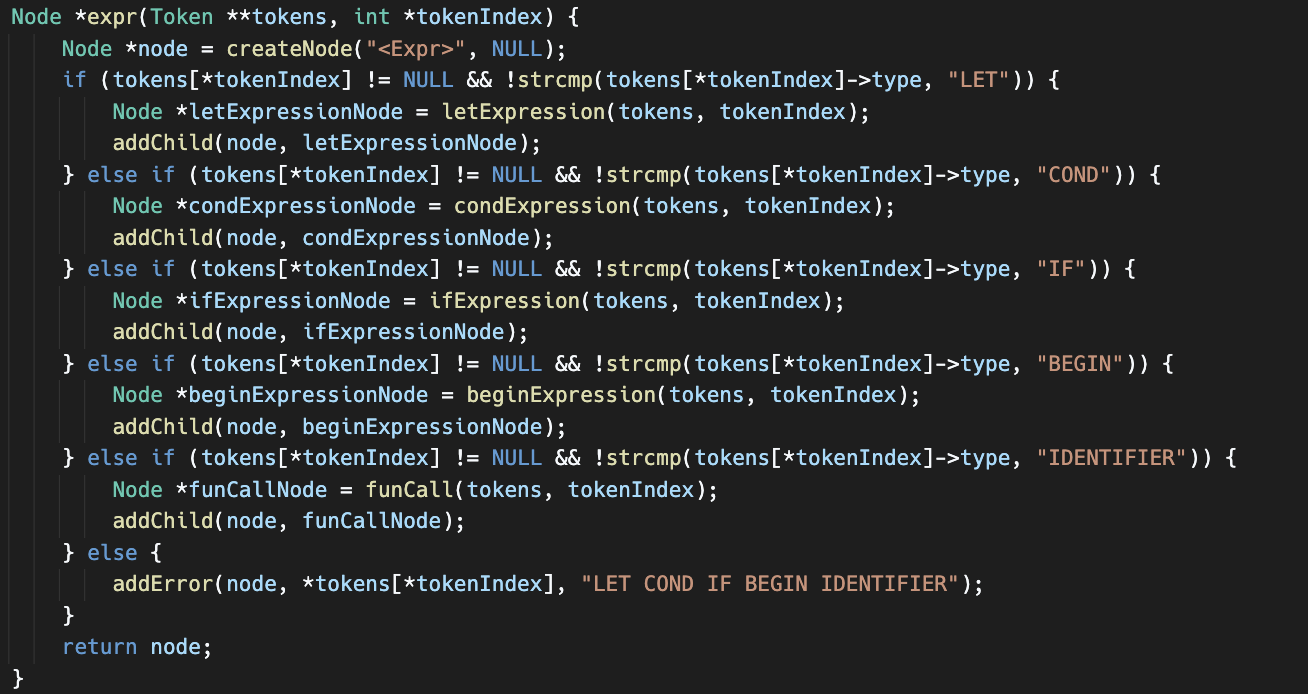
This function creates a node for the **<Statements>** type in the parse tree. It checks if the current token corresponds to an expression or a definition. If the token is an identifier, number, character, boolean, string, or left parenthesis, it calls the **expression** function to handle the expression and adds the resulting node as a child. If the token is "DEFINE", it calls the **definition** function to handle the definition and adds the resulting node as a child. The function also handles error cases when an unexpected token is encountered.



This function creates a node for the **<Expressions>** type in the parse tree. It first checks if the current token is a right parenthesis. If it is, it adds a placeholder node with "\_\_" as the value and returns the node. Otherwise, it calls the **expression** function to handle the expression and adds the resulting node as a child. Then, it recursively calls itself to handle the next expression in the list and adds the resulting node as a child. The function returns the created node.



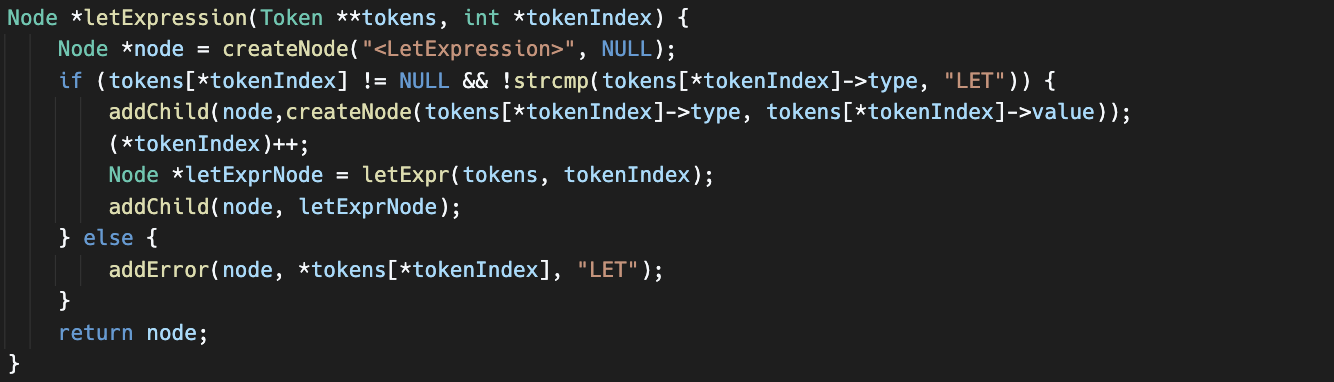
This function creates a node for the **<Expression>** type in the parse tree. It checks the type of the current token and handles different cases accordingly. If the token is an identifier, number, character, boolean, or string, it creates a node for that token and advances the token index. If the token is a left parenthesis, it creates the corresponding node and calls the **expr** function to handle the expression inside the parentheses. It then checks for a right parenthesis token and handles error cases accordingly.



This function creates a node for the **<Expr>** type in the parse tree. It checks the type of the current token and handles different cases accordingly. If the token is "LET", it calls the **letExpression** function to handle the let expression and adds the resulting node as a child. If the token is "COND", it calls the **condExpression** function to handle the cond expression and adds the resulting node as a child. If the token is "IF", it calls the **ifExpression** function to handle the if expression and adds the resulting node as a child. If the token is "BEGIN", it calls the **beginExpression** function to handle the begin expression and adds the resulting node as a child. If the token is an identifier, it calls the **funCall** function to handle the function call expression and adds the resulting node as a child. It also handles error cases when an unexpected token is encountered.



This function creates a node for a **<FunCall>** in the parse tree. It checks if the current token is of type "IDENTIFIER". If it is, it creates a node with the identifier value and advances the token index. Then, it calls the **expressions** function to parse the arguments of the function call and adds the resulting node as a child of the current node. If the token is not an identifier, it adds an error node to indicate the expected token type.



This function creates a node for the **< LetExpression>** in the parse tree. It checks if the current token is of type "LET". If it is, it creates a node with the token value and advances the token index. Then, it calls the **letExpr** function to parse the let expression body and adds the resulting node as a child of the current node. If the token is not "LET", it adds an error node to indicate the expected token type.

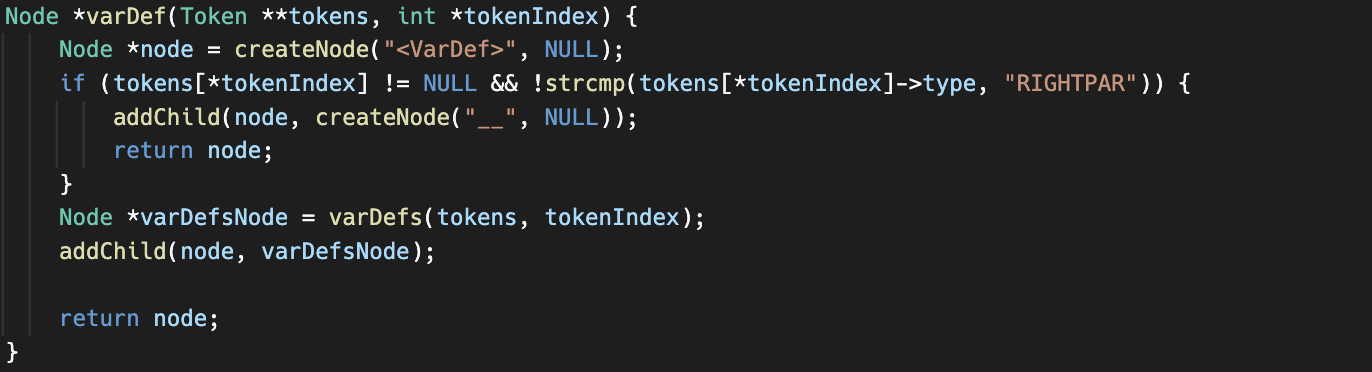


This function handles the parsing of a let expression body. It checks if the current token is of type "LEFTPAR". If it is, it creates a node with the token value and advances the token index. Then, it calls the **varDefs** function to parse the variable definitions within the let expression and adds the resulting node as a child of the current node. After that, it checks if the next token is of type "RIGHTPAR". If it is, it creates a node with the token value and advances the token index. Finally, it calls the **statements** function to parse the statements within the let expression body and adds the resulting node as a child of the current node. If the expected tokens are not found, it adds error nodes to indicate the expected token types.

A screen shot of a computer program

Description automatically generated with low confidence

This function handles the parsing of variable definitions within a **VarDefs** node in the parse tree. It checks if the current token is of type "LEFTPAR". If it is, it creates a node with the token value and advances the token index. Then, it checks if the next token is of type "IDENTIFIER". If it is, it creates a node with the token value and advances the token index. After that, it calls the **expression** function to parse the expression associated with the variable definition and adds the resulting node as a child of the current node. It then checks if the next token is of type "RIGHTPAR". If it is, it creates a node with the token value and advances the token index. Finally, it calls the **varDef** function to handle further variable definitions and adds the resulting node as a child of the current node. If the expected tokens are not found, it adds error nodes to indicate the expected token types.



This function handles the parsing of a variable definition within a **VarDef** node in the parse tree. It checks if the current token is of type "RIGHTPAR". If it is, it creates a node with a placeholder value (e.g., "\_\_") and returns the node. Otherwise, it calls the **varDefs** function to handle further variable definitions and adds the resulting node as a child of the current node. Finally, it returns the node.



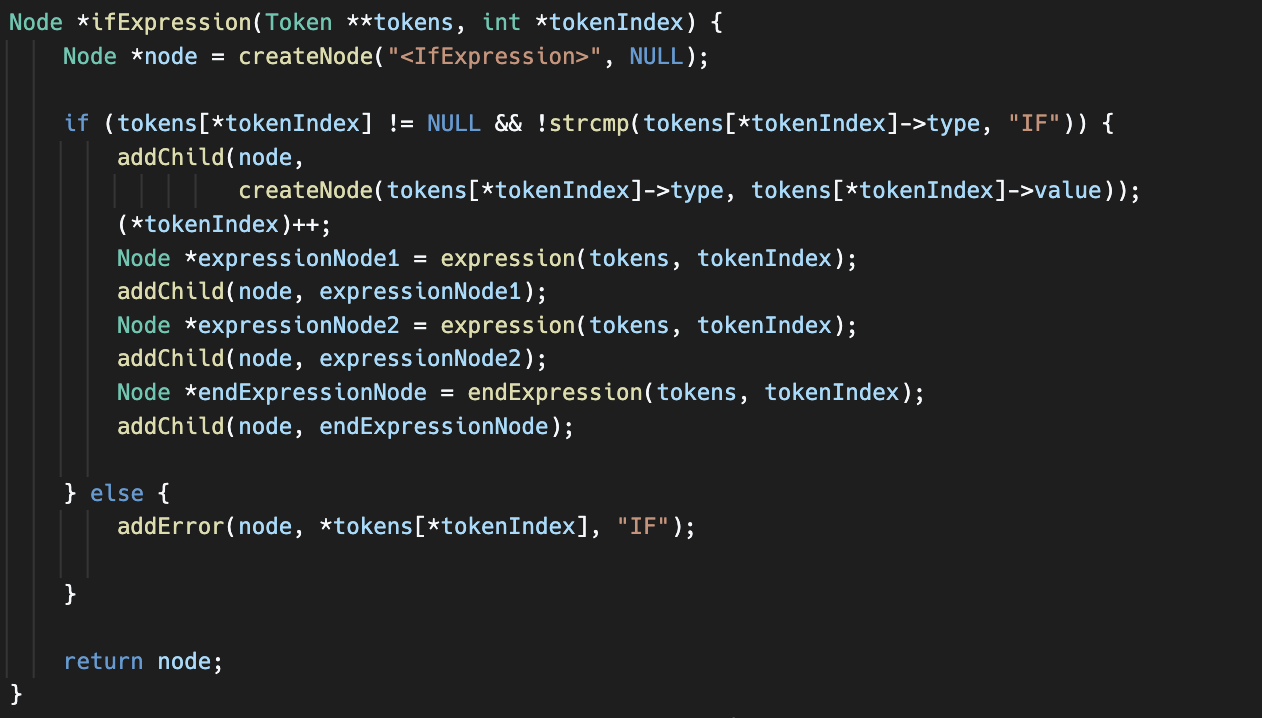
This function handles the parsing of a conditional expression within a **CondExpression** node in the parse tree. It checks if the current token is of type "COND". If it is, it creates a node with the token value and advances the token index. Then, it calls the **condBranches** function to parse the conditional branches and adds the resulting node as a child of the current node. Finally, it returns the node. If the expected token is not found, it adds an error node to indicate the expected token type.



This function handles the parsing of conditional branches within a **CondBranches** node in the parse tree. It checks if the current token is of type "LEFTPAR". If it is, it creates a node with the token value and advances the token index. Then, it calls the **expression** function to parse the condition expression and adds the resulting node as a child of the current node. Next, it calls the **statements** function to parse the statements within the conditional branch and adds the resulting node as a child of the current node. It then checks if the next token is of type "RIGHTPAR". If it is, it creates a node with the token value and advances the token index. Finally, it calls the **condBranch** function to handle additional conditional branches and adds the resulting node as a child of the current node. If the expected tokens are not found, it adds error nodes to indicate the expected token types.



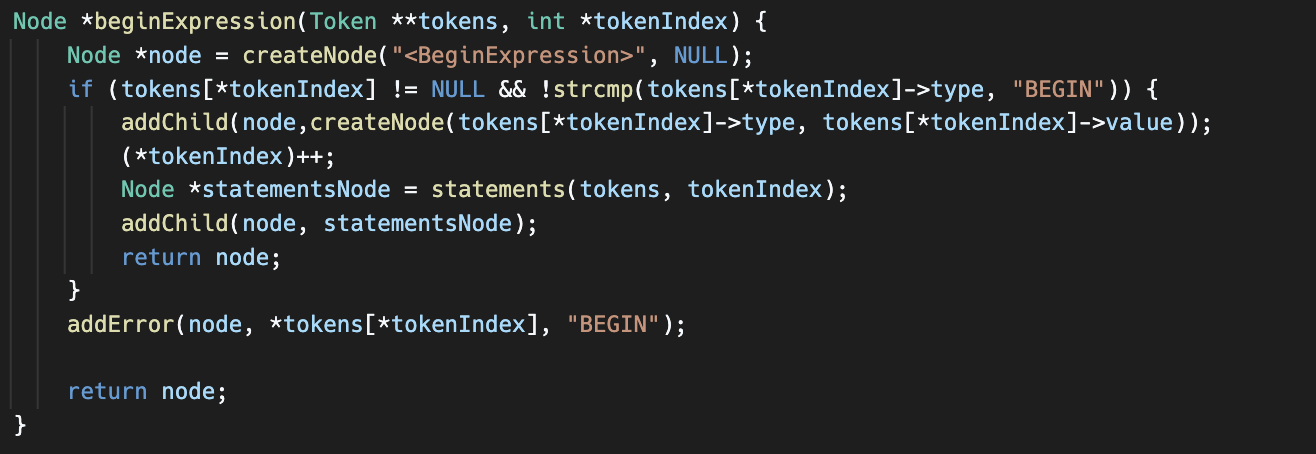
This function handles the parsing of a single conditional branch within a **CondBranch** node in the parse tree. It checks if the current token is of type "LEFTPAR". If it is, it creates a node with the token value and advances the token index. Then, it calls the **expression** function to parse the condition expression and adds the resulting node as a child of the current node. Next, it calls the **statements** function to parse the statements within the conditional branch and adds the resulting node as a child of the current node. It then checks if the next token is of type "RIGHTPAR". If it is, it creates a node with the token value and advances the token index. If the expected tokens are not found, it creates a placeholder node with a "\_\_" value. Finally, it returns the node.





This function handles the parsing of an if expression within an **IfExpression** node in the parse tree. It checks if the current token is of type "IF". If it is, it creates a node with the token value and advances the token index. Then, it calls the **expression** function to parse the condition expression and adds the resulting node as a child of the current node. Next, it calls the **expression** function again to parse the "then" expression and adds the resulting node as a child of the current node. Finally, it calls the **endExpression** function to parse the "end" expression and adds the resulting node as a child of the current node. If the expected token is not found, it adds an error node to indicate the expected token type.

This function handles the parsing of an end expression within an **EndExpression** node in the parse tree. It checks if the current token is of type "RIGHTPAR". If it is, it creates a placeholder node with a "\_\_" value and returns it. Otherwise, it calls the **expression** function to parse the expression and adds the resulting node as a child of the current node. Finally, it returns the node.



This function handles the parsing of a begin expression within a **BeginExpression** node in the parse tree. It checks if the current token is of type "BEGIN". If it is, it creates a node with the token value and advances the token index. Then, it calls the **statements** function to parse the statements within the begin expression and adds the resulting node as a child of the current node. Finally, it returns the node. If the expected token is not found, it adds an error node to indicate the expected token type.

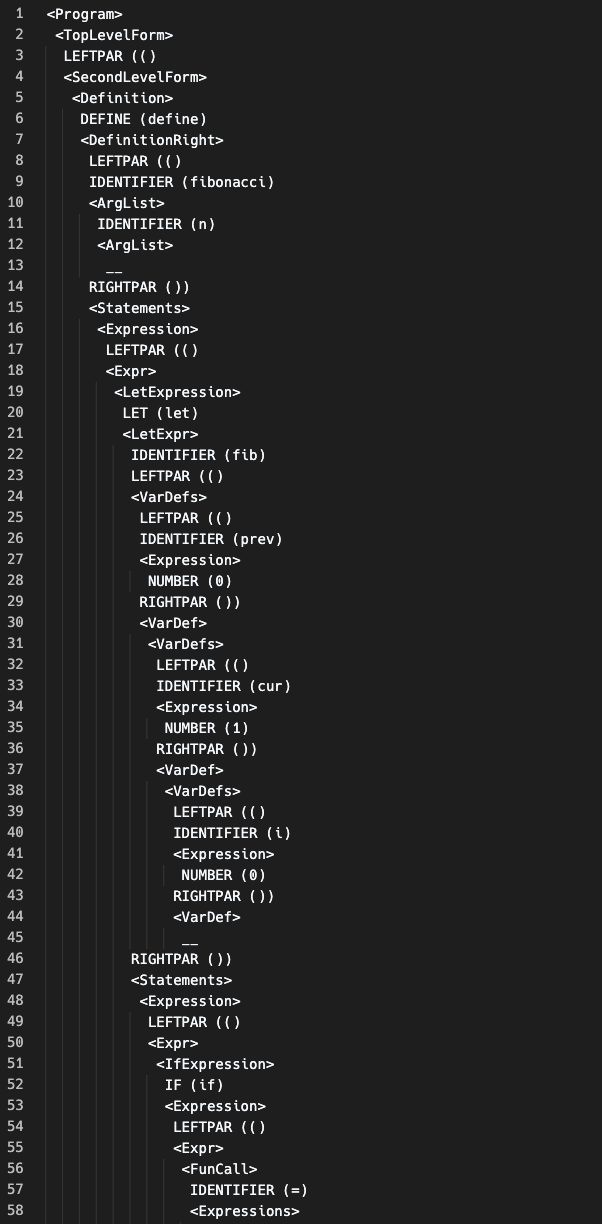
**INPUTS & OUTPUTS**

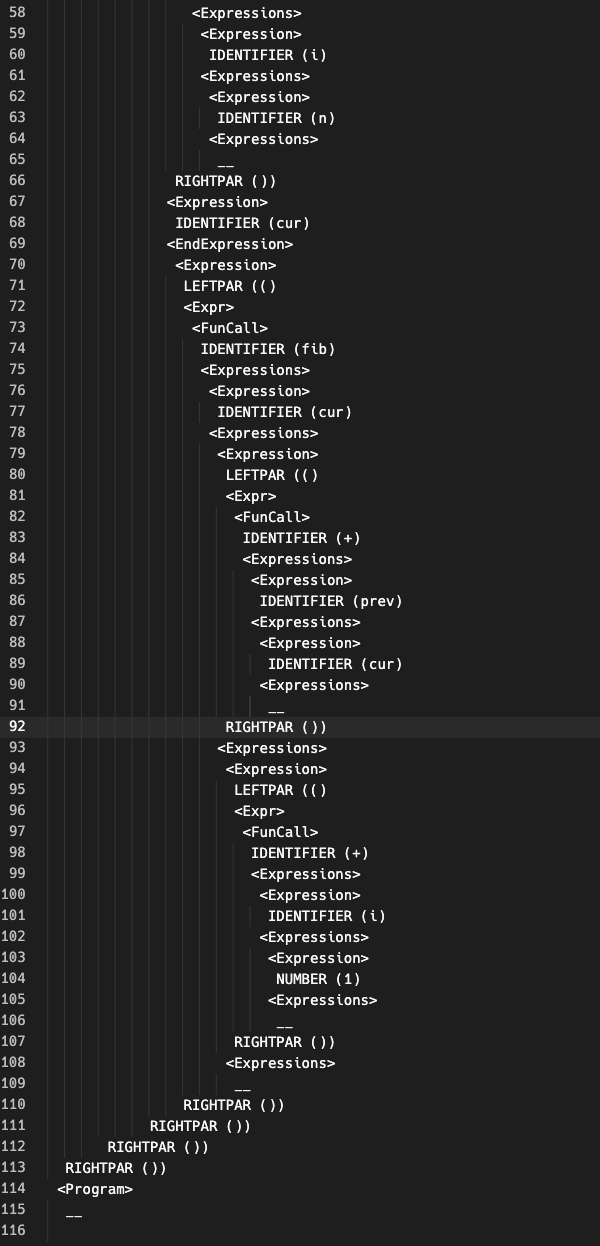
**Input-1**

A black background with white text

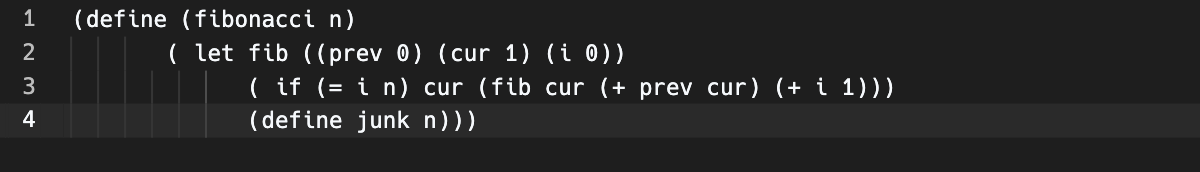
Description automatically generated with low confidence

**Output-1**





**Input-2**



**Output-2**

