T.C.

MANISA CELAL BAYAR UNIVERSITY

FACULTY OF ENGINEERING

DEPERTMANT OF COMPUTER ENGINEERING

ÇİZGİKOD

TECHNICAL REPORT

PREPARED BY

220315013 ALEYNA GÜLTEKİN

220315015 ESRA SEPİK

220315073 SİNEM TANRIVER

SUPERVISOR

DİDEM ABİDİN

MANISA 2025

1.INTRODUCTION

This report serves as a technical presentation of the "ÇİZGİKOD" project, undertaken as part of the Programming Languages course at Manisa Celal Bayar University's Department of Computer Engineering. The primary objective of this project is to design and implement a Turkish-based programming language, with its grammar defined using Backus-Naur Form (BNF).

During the language design process, we identified its fundamental features and developed lexical (tokenizer) and syntax (parser) analyzers responsible for parsing code to ensure its correct operation. These analyzers were developed by modifying the lexical analysis and parsing source codes provided in Chapter 4 of the textbook to suit our language design. The project implementation utilized Java, ensuring the source code compiles and runs.

Our report elaborates on the steps involved in our programming language design. The language's grammar is presented in BNF format. Additionally, a dedicated lookup table has been created for keywords and operators, incorporating Turkish characters. The report also includes state diagrams for important language structures such as if, for, and while. As part of the project, four example source codes (as separate text files) were generated in our designed programming language, demonstrating assignments, arithmetic expressions, and variable definitions. These example code files are successfully read by our modified lexical and syntax analysis codes, producing proper output.

2.PROJECT GOAL

The aim of this project is to design a Turkish-based programming language and define it using BNF (Backus-Naur Form). In designing the language, we will determine its fundamental features and create lexical (tokenizer) and syntax (parser) analyzers. These analyzers will be responsible for parsing the code written in the language to ensure its correct operation.

3. LOOK UP TABLE

İngilizce	Türkçe Karşılık
If	döfenşimos
Else	ornitorenk
While	pepe
For	bebe
Return	dede
Switch	huysuz
Case	uzun
Break	tontiş
Continue	şapşik
Input	marsupilami
Print	tospik
?	?
!	!
=	_
+	._.
-	,_,
*	*
/	\
:	:
;	;
^	*_*
%	%
Var	keloğlan
True	rik
False	morti
==	__
!=	!_
<	-:
>	:-
<=	-:_
>=	:-_
&	#
&&	##
|	$
||	$$
Int	mordekay
String	rigbi
Boolean	çakbeşlik
Float	finyıs
Char	förb
(	)
)	(
{	}
}	{
[	]
]	[
+=	._._

-= ,_,_

Start of Line ^_^

End of Line ^_^

4. BNF GRAMMER

<program> ::= <statement_list>

<statement_list> ::= <statement> | <statement> <statement_list>

<statement> ::= ^_^ <simple_statement> ^_^

| ^_^ <control_structure> ^_^

<simple_statement> ::= <declaration>

| <assignment>

| <expression>

| <print_statement>

| <input_statement>

| <return_statement>

| <break_statement>

| <continue_statement>

<declaration> ::= keloğlan <var_name> _ <expression>

<assignment> ::= <var_name> _ <expression>

<expression> ::= <arithmetic_expr>

| <comparison_expr>

| <boolean_expr>

| <var_name>

| <literal>

<arithmetic_expr> ::= <term> ((._. | ,_,) <term>)*

<term> ::= <factor> (\ <factor>)*

<factor> ::= <var_name>

| <literal>

|) <expression> (

<boolean_expr> ::= <expression> (# | ## | $ | $$) <expression>

<comparison_expr> ::= <expression> <comp_op> <expression>

<comp_op> ::= __ | !_ | -: | :- | -:_ | :-_

<literal> ::= <number> | <string> | <boolean_literal>

<number> ::= <int> | <float>

<int> ::= <digit>+

<float> ::= <digit>+ "." <digit>+

<string> ::= "\"" <char>* "\""

<boolean_literal> ::= rik | morti

<var_name> ::= <letter> <letter_or_digit>*

<letter_or_digit> ::= <letter> | <digit>

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<letter> ::= "A" | "B" | "C" | "Ç" | "D" | "E" | "F" | "G" | "Ğ" | "H" | "I" | "İ" | "J" | "K" |"L" |"M" |"N" |"O"

|"Ö" | "P" |"R" |"S" |"Ş" |"T" |"U"|"Ü" |"V" |"Y" |"Z"

"a" |"b" | "c" | "ç" | "d" | "e" | "f" | "g"|"ğ" | "h" | "ı" | "i" | "j" | "k" | ”l” |"m" | "n" | "o" | "ö" | "p" | "r" | "s" |"ş" | "t" | "u" | "ü" | "v" | "y" | "z"

<control_structure> ::= <if_structure>

| <while_structure>

| <for_structure>

| <switch_structure>

<if_structure> ::= döfenşimos) <comparison_expr> (} <statement_list> {

[ornitorenk } <statement_list> {]

tontiş

<while_structure> ::= pepe) <comparison_expr> (} <statement_list> { tontiş

<for_structure> ::= bebe) <assignment> ; <comparison_expr> ; <assignment> (} <statement_list> { tontiş

<switch_structure> ::= huysuz) <expression> (} <case_list> {

<case_list> ::= <case> | <case> <case_list>

<case> ::= uzun <literal> : } <statement_list> {

<return_statement> ::= dede <expression>

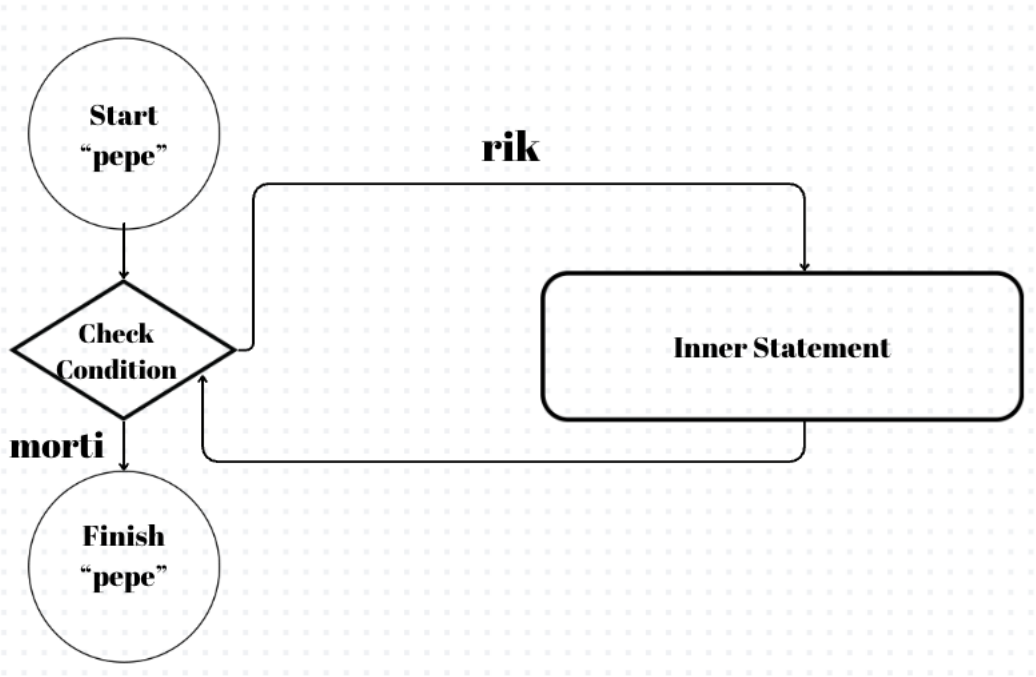
<break_statement> ::= tontiş

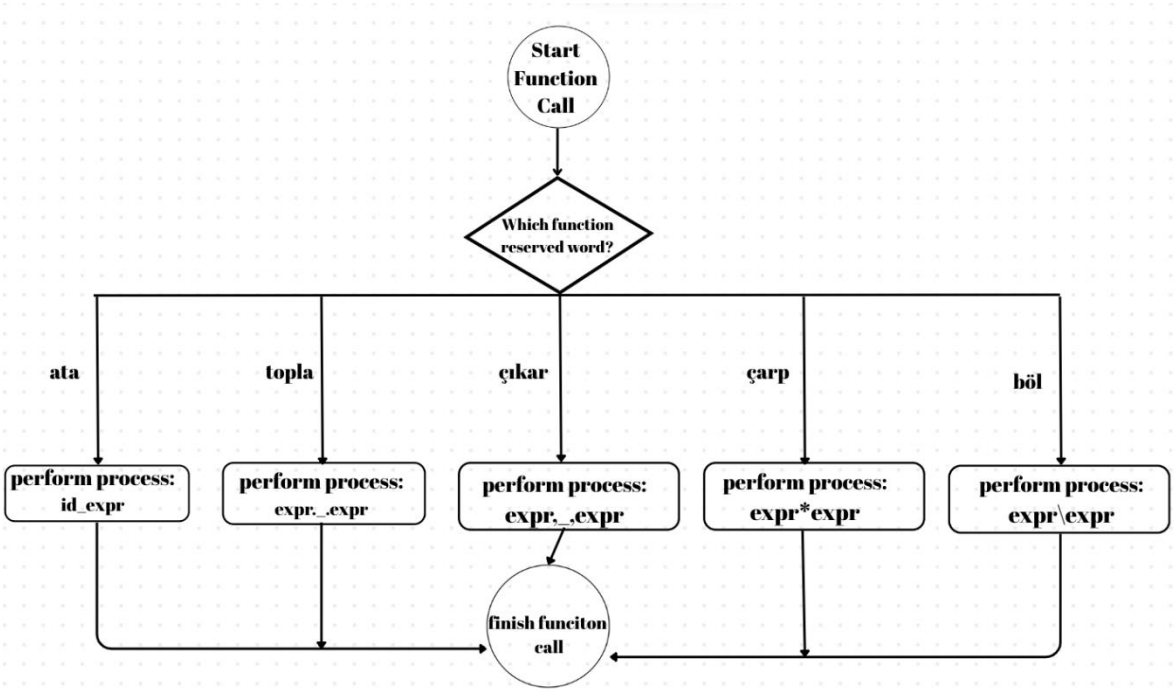
<continue_statement> ::= şapşik

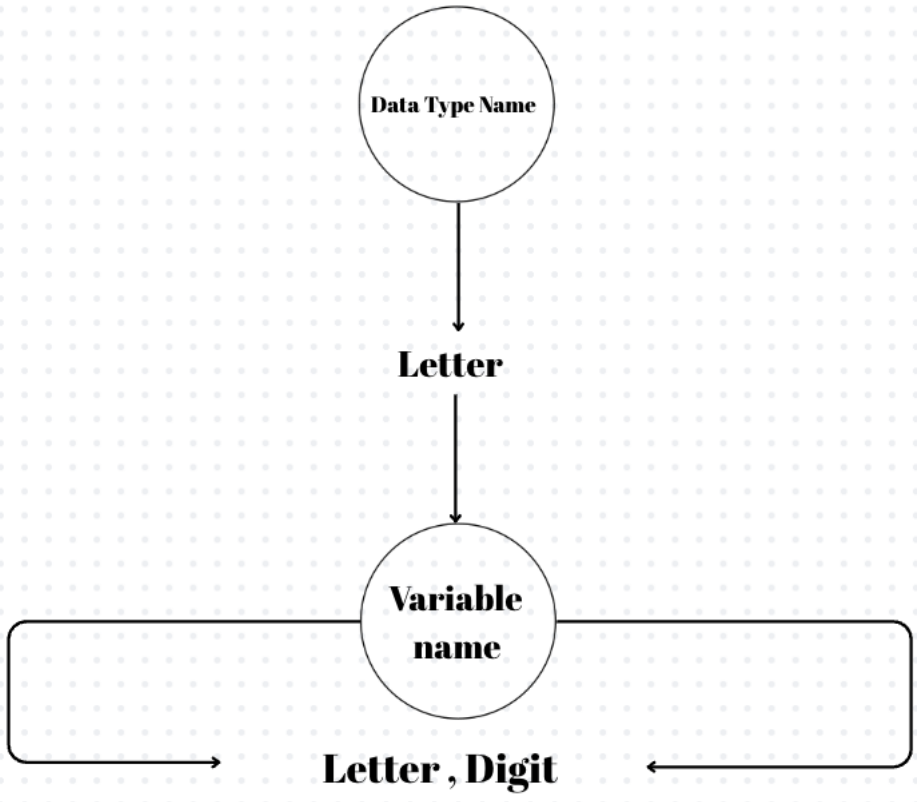
<input_statement> ::= marsupilami) <var_name> (

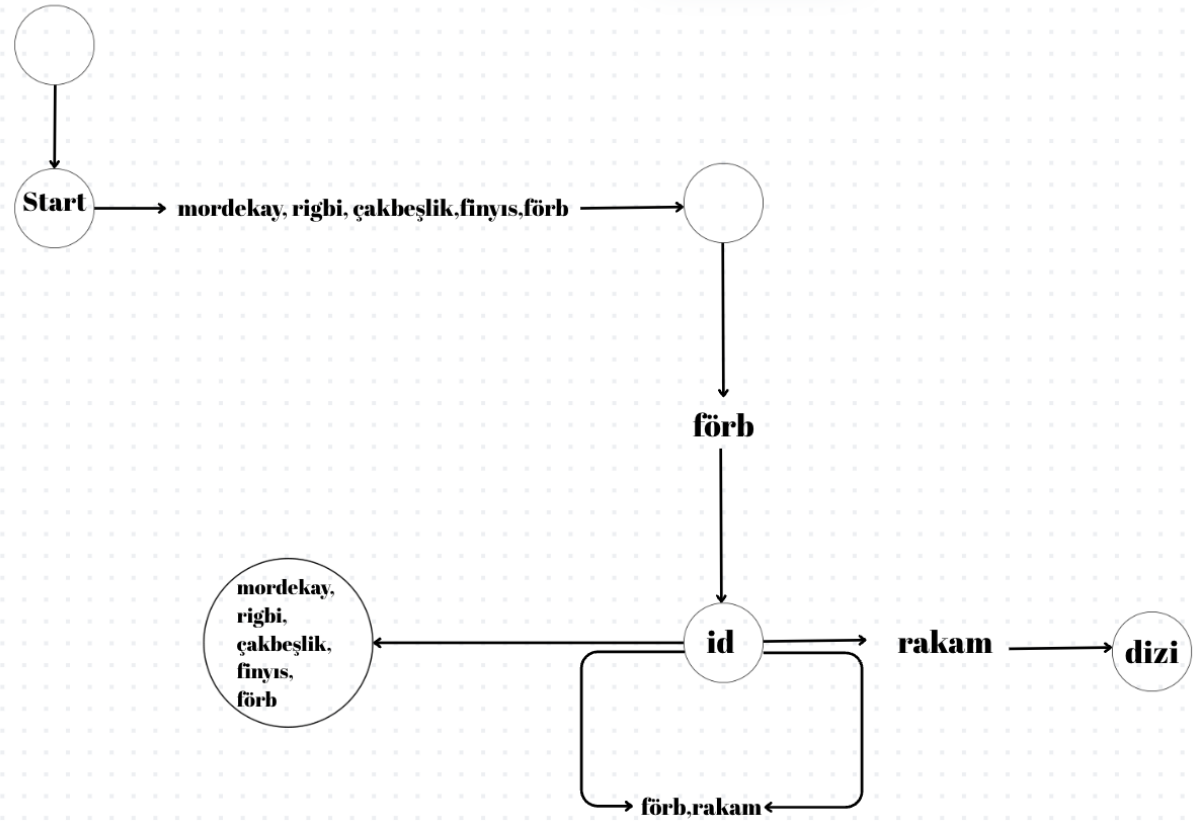
<print_statement> ::= tospik) <expression> (

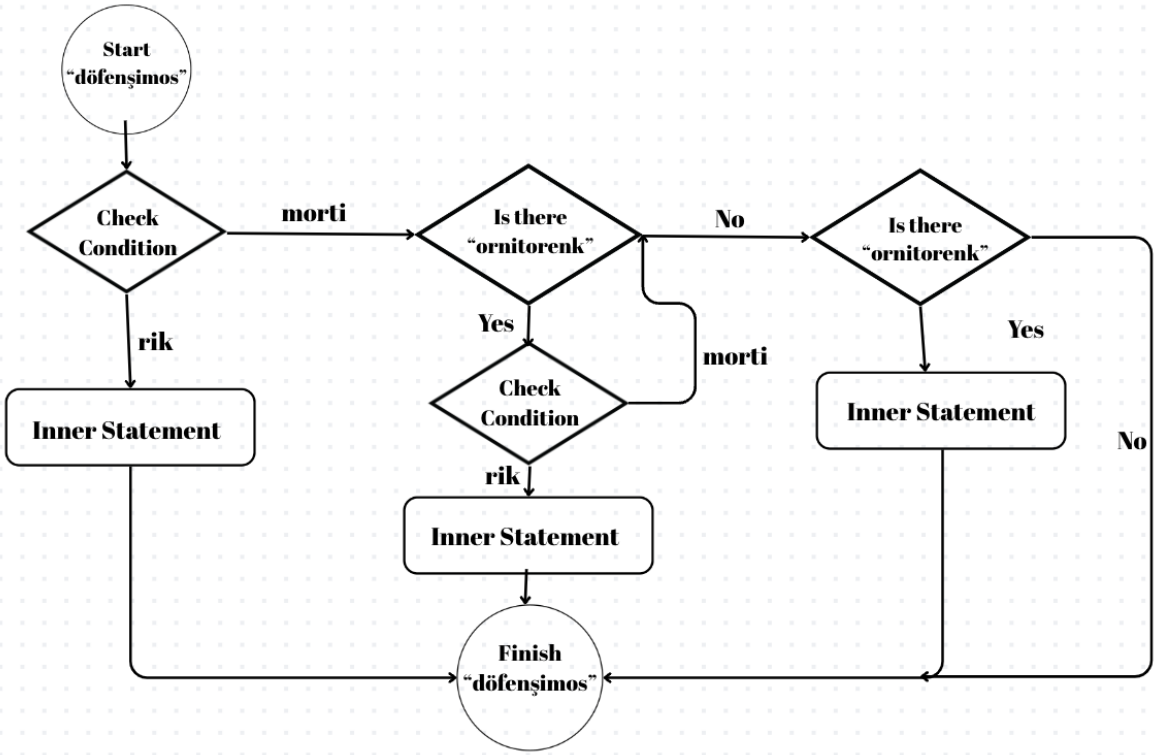
5.STATE DIAGRAMS




Start Function Call
Which function reserved word?
ata
topla
çıkar
çarp
böl
perform process: id_expr
perform process: expr._.expr
perform process: expr,_,expr
perform process: expr*expr
perform process: expr\expr
finish funciton call


Data Type Name
Letter
Variable name
Letter , Digit


Start
mordekay, rigbi, çakbeşlik,finyıs,förb
förb
mordekay,
rigbi,
çakbeşlik,
finyıs,
förb
id
rakam
dizi
förb,rakam


Start
"döfenşimos"
Check
Condition
morti
Is there
"ornitorenk"
No
Is there
"ornitorenk"
rik
Yes
morti
Yes
Check
Condition
Inner Statement
Inner Statement
rik
No
Inner Statement
Finish
"döfenşimos"

6.CLASSES

A.Purpose of the TokenType Class:

TokenType is an enum class used to define the type of each grammatical element (token) within the ÇizgiKod programming language. Its primary purpose is to specify the category to which words, symbols, and numbers recognized by the lexer (lexical analyzer) belong as it scans the source code.

This class acts as a bridge between the raw source code and the internal representation of the compiler/interpreter. By ensuring that each token (for instance, a keyword, an operator, a variable name, or a number) has a specific type, it enables the parser (syntactic analyzer) to correctly understand these tokens and build a structure according to the language's rules.

B.Purpose of the Token Class:

The Token class in the ÇizgiKod project is a fundamental data structure used to represent each lexical unit (token) identified during the lexical analysis (lexing) process. Its primary purpose is to store each meaningful part (keyword, operator, variable name, numerical value, text, etc.) detected by the lexer in the ÇizgiKod source code, in a standard format. This structure allows the parser (syntactic analyzer), which follows the lexer, to process the code more easily and systematically. Each Token object represents a single unit that carries meaning according to the language's grammatical rules.

Structure and Methods of the Token Class

The Token class has the following fields and methods:

Fields (Attributes):

TokenType type:

Purpose: Specifies the type of this token. It takes one of the values from the TokenType enum (e.g., IDENTIFIER, NUMBER, PLUS, VAR, etc.), which was explained previously. When the lexer recognizes a sequence of characters, it assigns the appropriate TokenType value to it.

String lexeme:

Purpose: This is the raw textual representation of the token as it appears in the source code. For example, for the keyword keloğlan, the lexeme would be "keloğlan"; for the number 123, it would be "123"; for the string literal "Merhaba", it would be ""Merhaba"".

Value value:

Purpose: This field is relevant only for tokens representing literal values such as numbers, strings, or booleans. It stores the parsed, actual value of the lexeme (e.g., Integer 123 for "123", String "hello" for ""hello""). For other token types (keywords, operators, identifiers), this field will be null.

Constructors:

public Token(TokenType type, String lexeme):

Purpose: Used to create tokens that are not immediate literal values (e.g., keywords, operators, identifiers). Since these tokens do not have a direct Value, the value field is set to null.

public Token(TokenType type, String lexeme, Value value):

Purpose: Used to create tokens that represent literal values like numbers, strings, or booleans. In this case, in addition to the lexeme, the parsed actual value of the lexeme (as a Value object) is also stored.

Accessor Methods (Getter Methods):

public TokenType getType():

Purpose: Returns the TokenType of the token. The parser uses this method to determine how to proceed based on the token's type.

public String getLexeme():

Purpose: Returns the raw text (lexeme) of the token as it appeared in the source code.

public Value getValue():

Purpose: Returns the parsed Value object of the token, if one exists. This is typically used when processing constant values like numbers or strings.

Other Methods:

@Override public String toString():

Purpose: Returns a string representation of the Token object. This method is primarily used for debugging purposes, allowing the content of tokens to be printed to the console. It includes the value field if it's not null, otherwise, it just shows the type and lexeme.

In summary, the Token class serves as the output of the lexer, the first phase of the ÇizgiKod compiler, and forms the input for the parser, the subsequent phase. Each Token object represents a small, meaningful, and easily processable unit of the source code.

C. Purpose of the Lexer Class

The Lexer (also known as a lexical analyzer or scanner) class in your ÇizgiKod project is a crucial component responsible for converting the source code into tokens (lexical units). Its main purpose is to read a line or a segment of the ÇizgiKod source code and break this raw text down into smaller, meaningful units called Token objects, which are understood by the language. It is the first phase in the compilation or interpretation process of a programming language. The Lexer identifies the language's defined keywords, operators, numbers, strings, and other symbols, converting each into the appropriate TokenType and Token objects.

Structure and Methods of the Lexer Class

Fields (Attributes):

private static final Map<String, TokenType> keywordMap:

Purpose: This static and final Map stores the mapping between all defined keywords and symbols (döfenşimos, _, ^_^, etc.) in the ÇizgiKod language and their corresponding TokenType enum values. It is initialized once within the static block and provides a quick lookup for the lexer when converting source code into tokens.

private static final Scanner scanner:

Purpose: This is a Scanner object used to get text input from the user via the console. This is necessary for situations like the marsupilami (INPUT) statement.

Static Initialization Block (static {}):

Purpose: This block runs once when the Lexer class is loaded into memory (typically when it's first used). It populates the keywordMap with all the special words and symbol-to-TokenType mappings of the ÇizgiKod language.

Methods:

public static List<Token> tokenize(String line):

Purpose: This is the core method of the Lexer class. It takes a String input (a line of source code) and converts this line into a list of Token objects.

How it works:

It splits the input line based on whitespace, while preserving text within double quotes (string literals) as a single part.

For each part, it performs the following checks in order:

Is it in keywordMap?: If the part is a keyword or a defined symbol (e.g., döfenşimos, _, +, etc.), it retrieves the corresponding TokenType from the map and creates a Token object. For rik (TRUE) and morti (FALSE), it specifically includes a Value object.

Is it a String Literal? (".*" regex): If the part is enclosed in double quotes, it removes the quotes and creates a Token of TokenType.STRING_LITERAL with a String Value.

Is it an Integer Literal? ([0-9]+ regex): If the part consists only of digits, it parses it into an Integer and creates a Token of TokenType.NUMBER with an Integer Value.

Is it a Float Literal? ([0-9]+\\.[0-9]+ regex): If the part resembles a decimal number, it parses it into a Double and creates a Token of TokenType.NUMBER with a Double Value.

Is it an Identifier? ([a-zA-Zçğıöşüİ_][a-zA-Z0-9çğıöşüİ_]* regex): If the part starts with a letter or underscore and is followed by letters, digits, or underscores, it recognizes it as an IDENTIFIER and creates a Token.

Unknown Token: If none of the above categories match, a Token of TokenType.UNKNOWN is created, indicating a lexical error.

It returns all the created tokens in a List<Token>.

public static String getUserInput():

Purpose: A simple helper method to get a single line of text input from the user. It prints a prompt to the console and returns the line entered by the user.

In summary, the Lexer class takes the raw text source code of a ÇizgiKod program and performs the critical first step of converting it into structured Token objects that the Parser class can then use to analyze the language's syntax.

D.Purpose of the Parser Class:

This code snippet enables the Parser class to correctly parse and evaluate arithmetic operations, handle operator precedence, and process fundamental syntactic units. It also includes helper methods that facilitate navigating the token list and reporting errors for the parser.

Explanation of the Methods

1. Arithmetic Operation Logic (Large IF block)

This block of code is most likely part of the parseTerm() method or a central helper method responsible for performing arithmetic operations.

Purpose: Performs arithmetic operations on given left and right Value objects according to a specific operator (e.g., DIVIDE, POWER, MOD). It supports different numerical data types (INT, FLOAT) and performs automatic type conversion (promoting int to float) when necessary.

Details:

Integer Operations (INT): If both operands are integers, it performs division, exponentiation, and modulo operations on integers. It checks for division by zero during division and modulo operations and throws an error.

Floating-Point Operations (FLOAT): If both operands are floating-point numbers, it performs the same operations on floating-point numbers, also including division by zero checks.

Mixed-Type Operations (INT and FLOAT): If one operand is an integer and the other is a floating-point number, it temporarily converts the integer to a floating-point number (double) (type promotion) and performs the operation on floating-point numbers. The result is always a floating-point number.

Error Handling: Throws a RuntimeException with an error message when encountering unsupported operand types.

Return Value: Returns a new Value object containing the result of the operation.

2. private Value parseFactor() throws RuntimeException

Purpose: Parses and evaluates the most fundamental and smallest units (factors) within expressions. These units typically include:

Literals: Numbers (NUMBER), strings (STRING_LITERAL), boolean values (TRUE, FALSE). It directly returns the Value of these tokens, which were pre-parsed by the Lexer.

Identifiers (IDENTIFIER): These are variable names. It retrieves and returns the value of the corresponding variable from the symbol table (symbolTable). It throws an error if an undefined variable is encountered.

Parenthesized Expressions (LPAREN Expression RPAREN): These are expressions enclosed within parentheses. In this case, it recursively calls the evaluateExpression() method to compute the value of the expression inside the parentheses. It also checks if the closing parenthesis (RPAREN) is correctly matched.

Error Handling: Throws a RuntimeException in cases of undefined variables or unexpected tokens.

Return Value: A Value object representing the evaluated factor's value.

3. private boolean match(TokenType type)

Purpose: A helper method that checks if the current token matches the expected TokenType.

How it works: If the current token's type is the same as the expected type, it increments the index to advance to the next token (i.e., "consumes" the token) and returns true. If they don't match, it returns false and does not change the index. The parser uses this method when it expects a specific syntactic structure.

4. private Token consume()

Purpose: Retrieves and returns the current token, then increments the index to advance to the next token.

How it works: It safely moves forward within the token list. If it attempts to consume tokens beyond the end of the input (if index exceeds the token list size), it throws an error. The parser uses this method when it has processed a token and needs to move to the next one.

5. private Token peek()

Purpose: "Peeks" at the current token, meaning it returns the token but does not advance the index (it does not "consume" the token).

How it works: It is typically used when the parser needs to look ahead at the next token to make a decision (e.g., checking what an if or while statement starts with). If the end of the token list is reached, it returns a dummy UNKNOWN (EOF) token.

6. private Token previousToken()

Purpose: Returns the token immediately preceding the currently processed token.

How it works: It returns the token at index - 1. This can be useful for error messages or in specific situations where information about the previously processed token is needed.

7. private Token currentToken()

Purpose: Returns the token at the current index position. It is similar to peek() but is generally used to directly retrieve the token currently being processed.

How it works: If the end of the token list is reached, it returns a dummy UNKNOWN (EOF) token.

8. private boolean error(String message)

Purpose: Used to report an error to the user when a parsing or execution failure occurs.

How it works: It prints the given error message to System.err (the standard error output stream). In addition to the message, it includes the lexeme and type of the current token (or "EOF" if at the end of input) to indicate where the error occurred. It always returns false to signal failure of the operation.

Collectively, these methods are the fundamental tools that enable the correct evaluation of expressions in the ÇizgiKod language and allow the parser to navigate the token stream effectively.

E. for the ÇizgiKodType enum and Value class:

The ÇizgiKodType enum and Value class in the ÇizgiKod project are fundamental building blocks for managing the language's data types and representing values within the program.

Purpose of the ÇizgiKodType Enum:

The ÇizgiKodType enum lists the different data types supported by the ÇizgiKod programming language. This enum is used to explicitly state which type of data a Value object or a variable holds. This allows the interpreter or compiler to perform operations on the correct data types and detect type mismatches.

Types included:

INT: Integer

STRING: String

BOOLEAN: Boolean value (true/false)

FLOAT: Floating-point number

CHAR: Character

VOID: Represents no return value (e.g., for certain statements or functions)

UNKNOWN: An unknown or undefined type (for errors or unexpected situations)

Purpose of the Value Class:

The Value class is a container (wrapper) class used to represent any value in the ÇizgiKod language (e.g., a variable's value, the result of an expression, a literal value). This class is vital for the interpreter to manage variable values and expression results along with their type information. Every value in ÇizgiKod is encapsulated as a Value object, allowing its type and actual data (as a Java Object) to be carried together.

Structure and Methods of the Value Class:

Fields (Attributes):

private ÇizgiKodType type:

Purpose: Specifies the ÇizgiKod language type of the data represented by this Value object (e.g., INT, STRING).

private Object data:

Purpose: Stores the actual data held by this Value object as a Java Object. For example, for an integer value, this field will contain an Integer object; for a string value, it will contain a String object.

Constructors:

public Value(ÇizgiKodType type, Object data):

Purpose: This is the main constructor for creating a Value object with a specified ÇizgiKodType and actual data.

public Value():

Purpose: Used specifically to create a Value object of VOID type. This can be used as a return value for statements that do not return a value (e.g., PRINT statements) or for functions. In this case, the data field is set to null.

Accessor and Manipulation Methods:

public ÇizgiKodType getType():

Purpose: Returns the ÇizgiKod data type (ÇizgiKodType) of this Value object.

public Object getData():

Purpose: Returns the raw Java Object data stored within this Value object.

public void setData(Object data):

Purpose: Used to update the data field of this Value object. This is particularly useful when assigning a new value to variables or writing the result of an expression to a variable.

Helper Conversion Methods (Type Casting Helpers):

These methods are used to safely convert the Object in the data field to specific Java types. This allows direct access to the relevant type in places like arithmetic and comparison operations within the Parser class.

public Integer asInt():

Purpose: Casts and returns the value in the data field as an Integer.

public Double asFloat():

Purpose: Casts and returns the value in the data field as a Double.

public String asString():

Purpose: Casts and returns the value in the data field as a String.

public Boolean asBoolean():

Purpose: Casts and returns the value in the data field as a Boolean.

Other Method:

@Override public String toString():

Purpose: Returns a human-readable string representation of the Value object. If the data field is null, it outputs "null (type)". Otherwise, it calls the toString() method of the data object itself to return its value as a string. This is useful for debugging and output.

In summary, while the ÇizgiKodType enum defines the language's type system, the Value class stores actual values according to these types and enables operations on them, thereby facilitating the runtime behavior of ÇizgiKod.

F. Purpose of the ÇizgiKod Class

The ÇizgiKod class orchestrates the execution of your project through its main method. It reads ÇizgiKod source code from external files (inputX.txt files), uses the Lexer to break down this code into tokens (lexemes), and then uses the Parser to interpret and run these tokens. This class acts as an orchestrator, demonstrating how the language works end-to-end.

Structure and Operation of the main Method:

The public static void main(String[] args) throws IOException method is where the program execution begins.

Looping Through Input Files (for (int i = 1; i <= 4; i++)):

The program is set up to process four separate input files, from input1.txt to input4.txt. This is ideal for testing different example ÇizgiKod programs according to your project requirements.

In each loop iteration, it reads the corresponding inputX.txt file and writes its output to an outputX.txt file.

File I/O Preparation (BufferedWriter, BufferedReader):

For each input file, a BufferedWriter is created to write the output.

A BufferedReader is used to read the input file. The try-with-resources statement ensures that both the reader and writer are automatically closed, even if an error occurs, improving resource management.

Initial Status Messages:

At the beginning of each program run, messages like "Program is compliant with the lookup table." and "Program started." are printed to both the output file and the console.

Lexical Analysis Phase (LEXICAL ANALYSIS PHASE):

The program reads each inputX.txt file line by line.

Lexer.tokenize(line): For every line read, the static tokenize method of the Lexer class is called. This method breaks down the line into ÇizgiKod tokens (e.g., keywords, operators, numbers, identifiers).

Token Collection: Tokens obtained from each line are collected into a single List<Token> called allTokens. This list is prepared to be fed as a single token stream to the parser.

Token Logging: Each line and the tokens derived from it are printed in detail to both the console and the outputX.txt file (including token code, lexeme, and value if present) for debugging and verification purposes.

Syntactic & Semantic Analysis / Interpretation Phase (SYNTACTIC & SEMANTIC ANALYSIS / INTERPRETATION PHASE):

After all tokens have been collected, a single Parser instance (new Parser(allTokens)) is created. This parser is used to check the program's syntax and to interpret/execute the ÇizgiKod code.

parser.parseProgram(): The main method of the Parser is called. This method parses the program and executes the statements within it.

Program Result: Based on the return value of the parseProgram() method, whether the program succeeded (without syntax or runtime errors) is reported to both the output file and the console.

Error Handling (try-catch blocks):

RuntimeException: Catches runtime errors from the parser (e.g., undefined variable, division by zero) and prints the corresponding error message.

FileNotFoundException: If one of the inputX.txt files is not found, an appropriate error message is provided to the user.

IOException: Catches other input/output errors (reading/writing issues).

finally block: Ensures that the BufferedWriter is properly closed even if an error occurs.

Program Termination Messages:

After each file is processed, a separator (----------) is printed.

After all files have been processed, a final message is printed indicating that the program has finished and that the outputX.txt files should be checked for detailed logs.

In summary, the ÇizgiKod class is the central control mechanism that brings together the Lexer and Parser components of the language, enabling the automated processing of ÇizgiKod programs and the logging of their results.

7.INPUTS

A. input1

^_^ keloğlan sayi _ 25 ^_^

^_^ tospik ("Başlangıç değeri: ") ^_^

^_^ tospik (sayi) ^_^

^_^ sayi .. 10 ^_^

^_^ tospik ("10 eklendi, yeni değer: ") ^_^

^_^ tospik (sayi) ^_^

^_^ döfenşimos (sayi :- 35) { ^_^

^_^ tospik ("Sayı 35 veya daha büyük.") ^_^

^_^ tontiş ^_^

} ^_^

^_^ tospik ("Programın sonuna gelindi.") ^_^

B. input2

^_^ keloğlan uzunluk _ 10 ^_^

^_^ keloğlan genişlik _ 5 ^_^

^_^ keloğlan alan _ 0 ^_^

^_^ alan _ uzunluk ._. genişlik ^_^

^_^ tospik) "Dikdörtgenin alanı: " ._. alan (^_^

^_^ tontiş ^_^

C. input3:

^_^ keloğlan yaş _ 10 ^_^

^_^ tospik) yaş (^_^

^_^ yaş _ 25 ^_^

^_^ tospik) yaş (^_^

^_^ tontiş ^_^

D. input4:

^_^ keloğlan 1sayi _ 20 ^_^ // Hata: Değişken adı sayı ile başlayamaz

^_^ tospik) "Merhaba" (^_^

^_^ keloğlan sonuc _ 5 $!$ 10 ^_^ // Hata: '$!$' tanınmayan bir operatör

^_^ döfenşimos) deger -: 10 (} ^_^

^_^ tospik) "Test" (^_^

^_^ tontiş ^_^

{ ^_^

8.OUTPUTS

Output - ÇizgiKod (run)

```
run:
Program started.

Processing Line 1: ^_^ keloğlan sayi _ 25 ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 22, Lexeme: keloğlan
  Next Token: 51, Lexeme: sayi
  Next Token: 14, Lexeme: _
  Next Token: 50, Lexeme: 25, Value: 25
  Next Token: 48, Lexeme: ^_^
----

Processing Line 2: ^_^ tospik ( "Başlangıç değeri: " ) ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 41, Lexeme: (
  Next Token: 49, Lexeme: "Başlangıç değeri: ", Value: Başlangıç değeri:
  Next Token: 40, Lexeme: )
  Next Token: 48, Lexeme: ^_^
----

Processing Line 3: ^_^ tospik ( sayi ) ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 41, Lexeme: (
  Next Token: 51, Lexeme: sayi
  Next Token: 40, Lexeme: )
  Next Token: 48, Lexeme: ^_^
----

Processing Line 4: ^_^ sayi ._. 10 ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 51, Lexeme: sayi
  Next Token: 15, Lexeme: ._.
  Next Token: 50, Lexeme: 10, Value: 10
  Next Token: 48, Lexeme: ^_^
----
```

Output - ÇizgiKod (run)

```
Processing Line 5: ^_^ tospik ( "10 eklendi, yeni değer: " ) ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 41, Lexeme: (
  Next Token: 49, Lexeme: "10 eklendi, yeni değer: ", Value: 10 eklendi, yeni değer:
  Next Token: 40, Lexeme: )
  Next Token: 48, Lexeme: ^_^
----

Processing Line 6: ^_^ tospik ( sayi ) ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 41, Lexeme: (
  Next Token: 51, Lexeme: sayi
  Next Token: 40, Lexeme: )
  Next Token: 48, Lexeme: ^_^
----

Processing Line 7: ^_^ döfenşimos ( sayi :-_ 35 ) { ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 1, Lexeme: döfenşimos
  Next Token: 41, Lexeme: (
  Next Token: 51, Lexeme: sayi
  Next Token: 30, Lexeme: :-_
  Next Token: 50, Lexeme: 35, Value: 35
  Next Token: 40, Lexeme: )
  Next Token: 43, Lexeme: {
  Next Token: 48, Lexeme: ^_^
----

Processing Line 8: ^_^ tospik ( "Sayı 35 veya daha büyük." ) ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 41, Lexeme: (
  Next Token: 49, Lexeme: "Sayı 35 veya daha büyük.", Value: Sayı 35 veya daha büyük.
  Next Token: 40, Lexeme: )
  Next Token: 48, Lexeme: ^_^
----
```

Output - ÇizgiKod (run)

```
Processing Line 9: ^_^ tontiş ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 8, Lexeme: tontiş
  Next Token: 48, Lexeme: ^_^
----

Processing Line 10: } ^_^
  Next Token: 42, Lexeme: }
  Next Token: 48, Lexeme: ^_^
----

Processing Line 11: ^_^ tospik ( "Programın sonuna gelindi." ) ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 41, Lexeme: (
  Next Token: 49, Lexeme: "Programın sonuna gelindi.", Value: Programın sonuna gelindi.
  Next Token: 40, Lexeme: )
  Next Token: 48, Lexeme: ^_^
----


--- SYNTACTIC & SEMANTIC ANALYSIS / INTERPRETATION PHASE ---

→ Değişken tanımlandı: sayi = 25
❌ Hata: Expected '(' after 'tospik'. (Satır: ( - Tip: RPAREN)
Program execution failed at statement.
❌ Program parsing or execution failed for input1.txt

----------------------------------------

Program started.

Processing Line 1: ^_^ keloğlan uzunluk _ 10 ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 22, Lexeme: keloğlan
  Next Token: 51, Lexeme: uzunluk
  Next Token: 14, Lexeme: _
  Next Token: 50, Lexeme: 10, Value: 10
```

Output - ÇizgiKod (run)

```
  Next Token: 50, Lexeme: 10, Value: 10
  Next Token: 48, Lexeme: ^_^
----

Processing Line 2: ^_^ keloğlan genişlik _ 5 ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 22, Lexeme: keloğlan
  Next Token: 51, Lexeme: genişlik
  Next Token: 14, Lexeme: _
  Next Token: 50, Lexeme: 5, Value: 5
  Next Token: 48, Lexeme: ^_^
----

Processing Line 3: ^_^ keloğlan alan _ 0 ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 22, Lexeme: keloğlan
  Next Token: 51, Lexeme: alan
  Next Token: 14, Lexeme: _
  Next Token: 50, Lexeme: 0, Value: 0
  Next Token: 48, Lexeme: ^_^
----

Processing Line 4:
----

Processing Line 5: ^_^ alan _ uzunluk ._. genişlik ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 51, Lexeme: alan
  Next Token: 14, Lexeme: _
  Next Token: 51, Lexeme: uzunluk
  Next Token: 15, Lexeme: ._.
  Next Token: 51, Lexeme: genişlik
  Next Token: 48, Lexeme: ^_^
----

Processing Line 6:
----
```

Output - ÇizgiKod (run)

```
Processing Line 7: ^_^ tospik ) "Dikdörtgenin alanı: " ._. alan ( ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 40, Lexeme: )
  Next Token: 49, Lexeme: "Dikdörtgenin alanı: ", Value: Dikdörtgenin alanı:
  Next Token: 15, Lexeme: ._.
  Next Token: 51, Lexeme: alan
  Next Token: 41, Lexeme: (
  Next Token: 48, Lexeme: ^_^
----

Processing Line 8: ^_^ tontiş ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 8, Lexeme: tontiş
  Next Token: 48, Lexeme: ^_^
----


--- SYNTACTIC & SEMANTIC ANALYSIS / INTERPRETATION PHASE ---

→ Değişken tanımlandı: uzunluk = 10
→ Değişken tanımlandı: genişlik = 5
→ Değişken tanımlandı: alan = 0
→ Değişken değeri güncellendi: alan = 15
Çıktı: Dikdörtgenin alanı: 15
→ 'tontiş' (break) ifadesi yürütüldü.
✔ Program parsed and executed successfully for input2.txt

----------------------------------------

Program started.

Processing Line 1: ^_^ keloğlan yaş _ 10 ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 22, Lexeme: keloğlan
  Next Token: 51, Lexeme: yaş
  Next Token: 14, Lexeme: _
  Next Token: 50, Lexeme: 10, Value: 10
```

Output - ÇizgiKod (run)

```
  Next Token: 48, Lexeme: ^_^
----

Processing Line 2: ^_^ tospik ) yaş ( ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 40, Lexeme: )
  Next Token: 51, Lexeme: yaş
  Next Token: 41, Lexeme: (
  Next Token: 48, Lexeme: ^_^
----

Processing Line 3: ^_^ yaş _ 25 ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 51, Lexeme: yaş
  Next Token: 14, Lexeme: _
  Next Token: 50, Lexeme: 25, Value: 25
  Next Token: 48, Lexeme: ^_^
----

Processing Line 4: ^_^ tospik ) yaş ( ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 40, Lexeme: )
  Next Token: 51, Lexeme: yaş
  Next Token: 41, Lexeme: (
  Next Token: 48, Lexeme: ^_^
----

Processing Line 5: ^_^ tontiş ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 8, Lexeme: tontiş
  Next Token: 48, Lexeme: ^_^
----


--- SYNTACTIC & SEMANTIC ANALYSIS / INTERPRETATION PHASE ---
```

Output - ÇizgiKod (run)

```
--- SYNTACTIC & SEMANTIC ANALYSIS / INTERPRETATION PHASE ---

→ Değişken tanımlandı: yaş = 10
Çıktı: 10
→ Değişken değeri güncellendi: yaş = 25
Çıktı: 25
→ 'tontiş' (break) ifadesi yürütüldü.
✔ Program parsed and executed successfully for input3.txt

----------------------------------------
```

Output - ÇizgiKod (run)

```
Program started.

Processing Line 1: ^_^ keloğlan 1sayi _ 20 ^_^  // Hata: Değişken adı sayı ile başlayamaz
  Next Token: 48, Lexeme: ^_^
  Next Token: 22, Lexeme: keloğlan
  Next Token: 99, Lexeme: 1sayi
  Next Token: 14, Lexeme: _
  Next Token: 50, Lexeme: 20, Value: 20
  Next Token: 48, Lexeme: ^_^
  Next Token: 99, Lexeme: //
  Next Token: 99, Lexeme: Hata:
  Next Token: 51, Lexeme: Değişken
  Next Token: 51, Lexeme: adı
  Next Token: 51, Lexeme: sayı
  Next Token: 51, Lexeme: ile
  Next Token: 51, Lexeme: başlayamaz
----

Processing Line 2: ^_^ tospik ) "Merhaba" ^_^ // Hata: 'tospik' fonksiyonunun parantezi eksik
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 40, Lexeme: )
  Next Token: 49, Lexeme: "Merhaba", Value: Merhaba
  Next Token: 48, Lexeme: ^_^
  Next Token: 99, Lexeme: //
  Next Token: 99, Lexeme: Hata:
  Next Token: 99, Lexeme: 'tospik'
  Next Token: 51, Lexeme: fonksiyonunun
  Next Token: 51, Lexeme: parantezi
  Next Token: 51, Lexeme: eksik
----

Processing Line 3: ^_^ keloğlan sonuc _ 5 $!$ 10 ^_^ // Hata: '$!$' tanınmayan bir operatör
  Next Token: 48, Lexeme: ^_^
  Next Token: 22, Lexeme: keloğlan
  Next Token: 51, Lexeme: sonuc
  Next Token: 14, Lexeme: _
  Next Token: 50, Lexeme: 5, Value: 5
  Next Token: 99, Lexeme: $!$
```

Output - ÇizgiKod (run)

```
  Next Token: 50, Lexeme: 5, Value: 5
  Next Token: 99, Lexeme: $!$
  Next Token: 50, Lexeme: 10, Value: 10
  Next Token: 48, Lexeme: ^_^
  Next Token: 99, Lexeme: //
  Next Token: 99, Lexeme: Hata:
  Next Token: 99, Lexeme: '$!$'
  Next Token: 51, Lexeme: tanınmayan
  Next Token: 51, Lexeme: bir
  Next Token: 51, Lexeme: operatör
----

Processing Line 4: ^_^ döfenşimos ) deger -: 10 ( } ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 1, Lexeme: döfenşimos
  Next Token: 40, Lexeme: )
❌ Hata: Expected identifier after 'keloğlan'. (Satır: 1sayi - Tip: UNKNOWN)
  Next Token: 51, Lexeme: deger
  Next Token: 27, Lexeme: -:
Program execution failed at statement.
  Next Token: 50, Lexeme: 10, Value: 10
  Next Token: 41, Lexeme: (
❌ Program parsing or execution failed for input4.txt
  Next Token: 42, Lexeme: }
  Next Token: 48, Lexeme: ^_^
----

Processing Line 5: ^_^ tospik ) "Test" ( ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 40, Lexeme: )
  Next Token: 49, Lexeme: "Test", Value: Test
  Next Token: 41, Lexeme: (
  Next Token: 48, Lexeme: ^_^
----

Processing Line 6: ^_^ tontiş ^_^
  Next Token: 48, Lexeme: ^_^
```

Output - ÇizgiKod (run)

```
  Next Token: 27, Lexeme: -:
Program execution failed at statement.
  Next Token: 50, Lexeme: 10, Value: 10
  Next Token: 41, Lexeme: (
❌ Program parsing or execution failed for input4.txt
  Next Token: 42, Lexeme: }
  Next Token: 48, Lexeme: ^_^
----

Processing Line 5: ^_^ tospik ) "Test" ( ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 11, Lexeme: tospik
  Next Token: 40, Lexeme: )
  Next Token: 49, Lexeme: "Test", Value: Test
  Next Token: 41, Lexeme: (
  Next Token: 48, Lexeme: ^_^
----

Processing Line 6: ^_^ tontiş ^_^
  Next Token: 48, Lexeme: ^_^
  Next Token: 8, Lexeme: tontiş
  Next Token: 48, Lexeme: ^_^
----

Processing Line 7: } ^_^
  Next Token: 42, Lexeme: }
  Next Token: 48, Lexeme: ^_^
----


--- SYNTACTIC & SEMANTIC ANALYSIS / INTERPRETATION PHASE ---


------------------------------------------

Program finished. Check outputX.txt files for detailed logs and console for ÇizgiKod output.
BUILD SUCCESSFUL (total time: 1 second)
```