FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT
2024-2025 SPRING

CSE2118 Programming Languages
Term Project

Student ID	Name Surname
230315086	Rafet TAŞKIN
220315044	Mert YÜKSEL
220315049	Ahmet Alp ÖZDEMİR
220315046	Muhammet Sencer DEDE

TABLE OF CONTENTS

TABLE OF CONTENTS **2**
1. Introduction **3**
2. Language Overview **3**
3. Grammar and Syntax **4**
3.1 Backus-Naur Form (BNF) 4
3.2 Look-up Table and Keywords 7
Keywords 7
4. Control Structures **8**
4.1 If statement(şayet): 9
4.2 While Loop (madem): 10
4.3 Semantic Analysis **10**
4.4 Error Handling at Front end **10**
5. Virtual Machine Architecture **11**
5.1 Core Components 11
5.2 Execution Model 11
6. Bytecode Specification **12**
6.1 Bytecode Instruction Set 12
6.2 Binary Operations 13
6.3 Unary Operations 14
7. Memory Management **14**
7.1 MEObject* 15
7.2 Stack Management 16
7.3 Object Representation 16
8. Execution Example **17**
8.1 Sample Program 17
8.2 Compilation to Bytecode 17
10. Building Manisa Engereği and running a program written in Manisa Engereği **18**
11. Possible Improvements **19**
12. More Code examples **19**

(https://github.com/rft0/manisa-engeregi)

1. Introduction

Manisa Engereği is a general-purpose, high-level programming language designed with two goals in mind: simplicity and humor. While it serves as a parody of traditional languages, it is also a fully functional system — complete with its own bytecode compiler, and virtual machine.

The language is intentionally informal and playful, but this does not come at the cost of technical depth. Manisa Engereği strikes a unique balance between being entertaining and

operational. Its humorous nature is most evident in its Turkish-language keywords, which parody common programming constructs while remaining syntactically sound. These will be explained in detail later in the report.

2. Language Overview

Manisa Engereği is a dynamically typed language, meaning that variable types are not explicitly declared by the programmer. Instead, all variables are defined using the keyword **değişken or sabit**, which is the Turkish word for "variable" and “constant”. All variables in Manisa Engereği, even built-in functions or user defined functions in code all are objects; this approach makes language much easier to use.

This design choice reflects the language's emphasis on **simplicity** and **minimalism**. In keeping with this philosophy, certain commonly used control structures such as for **loops** and **switch** statements have been deliberately omitted. We believe that the **while** loop provides sufficient expressive power for all forms of iteration, from basic counting to complex conditional loops. Similarly, all conditional branching in Manisa Engereği is handled using if and else constructs. The absence of switch statements not only reduces the number of language constructs that need to be learned, but also reinforces a consistent and uniform style of decision-making in code.

By removing syntactic noise and reducing feature overload, Manisa Engereği encourages developers to focus on the logic and flow of their programs rather than getting bogged down in language-specific quirks. This minimalism also aligns with the humorous and light-hearted spirit of the language — it does not try to imitate mainstream languages

feature-for-feature, but instead embraces a simpler and more accessible approach to programming.

3. Grammar and Syntax

This section provides a formal description of the grammar and syntax of the Manisa Engereği programming language. The structure of valid programs is defined using **Backus-Naur Form (BNF).** These rules define how programs are written, how expressions and statements are formed, and how different language constructs interact.

And a **look-up table** of keywords is also provided. This table includes all reserved words and symbols used in Manisa Engereği, along with their meanings and roles within the language. Many of these keywords are humorous or culturally inspired Turkish phrases, intentionally chosen to enhance the language's playful tone while still maintaining logical functionality.

3.1 Backus-Naur Form (BNF)

```
<program> ::= <stmt>*


<stmt> ::= <if_stmt>
| <while_stmt>
| <block>
| <method_decl>
| <return_stmt>
| <break_stmt>
| <continue_stmt>
| <expr_stmt>


<expr_stmt> ::= <expr> ";"
<if_stmt> ::= "şayet" "(" <expr> ")" <stmt> ["değilse" <stmt>]
```

```
<while_stmt> ::= "madem" "(" <expr> ")" <stmt>

<block> ::= "{" <stmt>* "}"

<method_decl> ::= "marifet" IDENTIFIER "(" [<parameters>] ")" <block>

<parameters> ::= IDENTIFIER ("," IDENTIFIER)*

<return_stmt> ::= "tebliğ" [<expr>] ";"

<break_stmt> ::= "yeter" ";"

<continue_stmt> ::= "devam" ";"



# Expressions

<expr> ::= <assignment>

<assignment> ::= <logical_or> ( ("=" | "+=" | "-=" | "*=" | "/=" | "%=" |
"&=" | "|=" | "^=" | "~=") <assignment> )?

<logical_or> ::= <logical_and> (<logical_or_operator> <logical_and>)*

<logical_or_operator> ::= "||" | "yahut"

<logical_and> ::= <equality> (<logical_and_operator> <equality>)*

<logical_and_operator> ::= "&&" | "ve"

<equality> ::= <comparison> (("==" | "!=") <comparison>)*

<comparison> ::= <bitwise_or> (("<" | "<=" | ">" | ">=") <bitwise_or>)*

<bitwise_or> ::= <bitwise_xor> ("|" <bitwise_xor>)*

<bitwise_xor> ::= <bitwise_and> ("^" <bitwise_and>)*

<bitwise_and> ::= <shift> ("&" <shift>)*

<shift> ::= <term> (("<<" | ">>") <term>)*

<term> ::= <factor> (("+" | "-") <factor>)*

<factor> ::= <unary> (("" | "/" | "%") <unary>)
```

```
<unary> ::= ("-" | "!" | "~" | "gayrı") <unary>

| <postfix>

<postfix> ::= <primary> ("++" | "--")?

<primary> ::= IDENTIFIER

| IDENTIFIER "(" [<arguments>] ")"

| INTEGER

| FLOAT

| STRING

| "(" <expr> ")"

<arguments> ::= <expr> ("," <expr>)*

# Terminals (Tokens)

IDENTIFIER ::= [a-zA-Z_][a-zA-Z0-9_]*

INTEGER ::= [0-9]+

FLOAT ::= [0-9]+.[0-9]+

STRING ::= """ .* """

NEWLINE ::= ";"

COMMENT ::= # .* NEWLINE
```

As can be seen in the BNF definitions, Manisa Engereği uses **curly braces {}** to define code blocks and structure control flow. Rather than relying on indentation (like Python), this approach provides a more explicit and familiar block structure, similar to languages like C, Java, and JavaScript.

In addition, **statements are terminated using a semicolon ;**, which serves as the newline or end-of-instruction marker. This, too, mirrors the syntax of C-style languages.

3.2 Look-up Table and Keywords

The Manisa Engereği language supports Turkish characters in identifiers and includes Turkish-language keywords. Below is the look-up table that defines the language's character classes and all reserved keywords.

Class	Characters / Description
Letters	a–z, A–Z, **ğ, Ğ, ü, Ü, ş, Ş, ö, Ö, ç, Ç, ı, İ**
Digits	0–9
Operators	+, -, *, /, =, ==, !=, <, <=, >, >=
Delimiters	(,), {, }, ;, ,
String Quote	“”
Whitespace	Space, tab, newline
Comments	# for single-line comments

Keywords

Other Languages	Manisa Engereği
const	sabit
let, variable	değişken
and	ve
or	yahut
if	şayet
else	değilse
while	madem
method, function	marifet
return	tebliğ
break	yeter

`continue`	`devam`
`none`	`yok`
`not`	`gayrı`

4. Control Structures

Manisa Engereği supports only two control structures: the **if** statement and the **while** loop. These constructs are fundamental to nearly all programming languages, and they are sufficient for expressing most conditional logic and repetition patterns. In line with the language's minimalist philosophy, we intentionally excluded other control structures such as for loops and switch cases, as discussed in the previous section.

The keywords used for control structures in Manisa Engereği reflect the language's humorous and culturally specific character:

şayet — used to introduce an if condition

değilse — used for the else clause

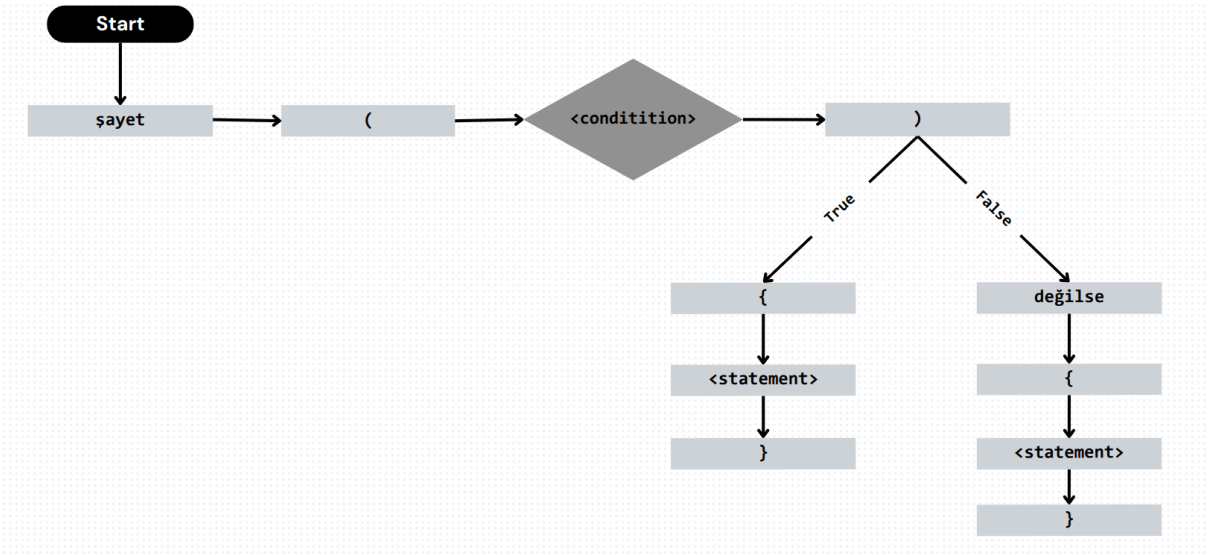
değilse and şayet can be used respectively to create an else if block.

madem — used to begin a while loop

These keywords are both functional and playful, aiming to bring a smile to Turkish-speaking programmers while remaining clear in their intent.

Note: Manisa engereği can parse and interpret complex expressions like assignments / compound assignments in conditions.

4.1 If statement(şayet):



For example:

```
şayet (x > 10) {
    tebliğ "büyükmüş";
} değilse {
    tebliğ "küçükmüş";
}
```

Another example with logical operations:

```
şayet ((x > 9) ve (x < 100)) {
      çıktı("Sayı iki basamaklı"):
}
```

And also it is equivalent to

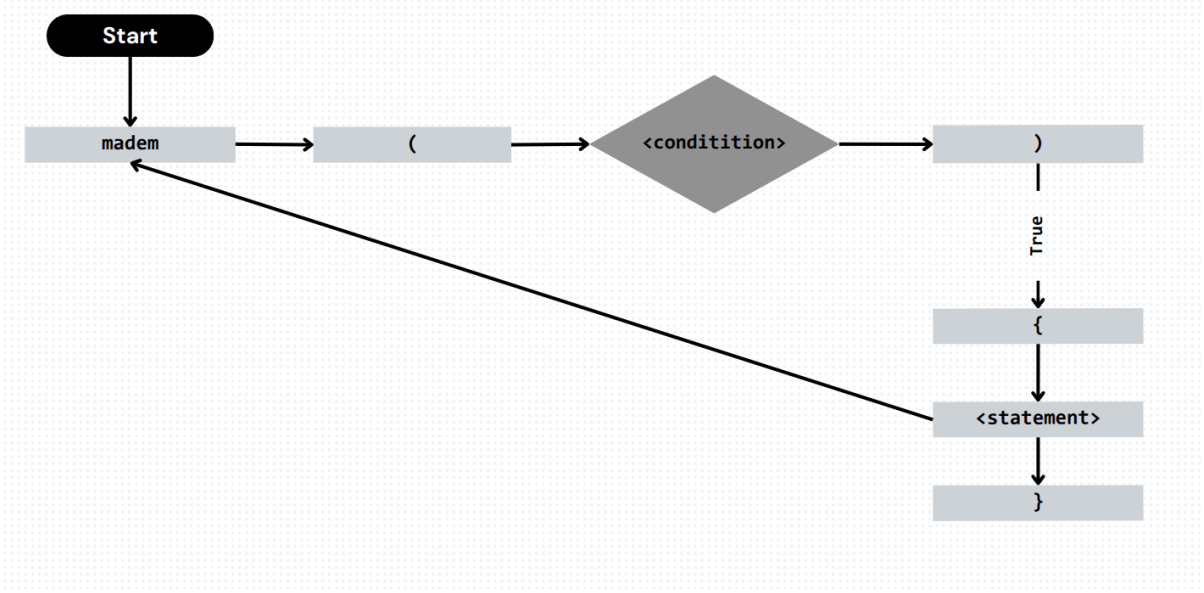
```
şayet ((x > 9) && (x < 100)) {
      çıktı("Sayı iki basamaklı"):
}
```

With a logical or example

```
şayet ((x == 0) yahut (x == -1)) {
      tebliğ;
}
```

(**Note:** In manisa engereği every type object have tp_bool implemented ie: an empty string ("") returns false always, if a float or integer not equal to 0 it is true always etc.)

4.2 While Loop (madem):



For example:

```
// Prints “I love Manisa Engereği” 5 times
değişken sayı = 0;
madem (sayı < 5) {
    çıktı("I love Manisa Engereği");
    sayı = sayı + 1;
}
```

4.3 Semantic Analysis

When lexing / parsing ends, manisa engereği analyses parser output to check if there are any undefined variables / functions, anything that violates variables “değişken”, “sabit” state and many more things. Found errors / warnings are added as diagnostics.

4.4 Error Handling at Front end

When lexer encounters an error it doesn’t halt code progress immediately instead it recovers the error and continues checking errors. Same goes for parser when parser encounters with error it doesn’t halt immediately it continues to parse statements. If there are errors at the lexing, parsing and analysing ends, the diagnostics system throws info about all errors / warnings at once when parsing ends.

5. Virtual Machine Architecture

The Manisa Engereği virtual machine (VM) is a stack-based interpreter that executes compiled bytecode.

5.1 Core Components

The VM consists of several key components:

- **Instruction Pointer (IP):** Tracks the current bytecode instruction being executed
- **Stack Pointer (SP):** Points to the top of the stack
- **Stack:** Stores **MEObject*** pointers.
- **Constant Pool:** Contains all literal values used in the program
- **Global Variables:** Stores all global variable references
- **Local Variables:** Stores variables local to the current scope

5.2 Execution Model

The VM follows a simple execution cycle:

1. Fetch the next bytecode operation
2. Decode the operation
3. Execute the operation
4. Repeat until program completion or error

This cycle continues until either:

- All bytecode instructions have been executed
- A tebliğ (return) instruction is encountered
- An error condition occurs

6. Bytecode Specification

The Manisa Engereği bytecode is a compact, linear sequence of instructions that represent the compiled program. Each instruction consists of:

- A 1-byte opcode
- Variable operand bytes (depends on op)

6.1 Bytecode Instruction Set

The current implementation supports the following core instructions (from the source code of Manisa Engereği language):

```
typedef enum {
    CO_OP_NOP,
    CO_OP_LOAD_CONST,
    CO_OP_LOAD_GLOBAL,
    CO_OP_LOAD_VARIABLE,
    CO_OP_STORE_GLOBAL,
    CO_OP_STORE_VARIABLE,
    CO_OP_BINARY_OP,
    CO_OP_UNARY_OP,
    CO_OP_CALL_FUNCTION,
    CO_OP_RETURN,
    CO_OP_POP,
    CO_OP_JUMP_REL,
    CO_OP_JUMP_IF_FALSE,
} MECodeOp;
```

The following table provides a detailed explanation of each instruction set:

Opcode	Name	Description	Operands
CO_OP_NOP	No operation	Nothing, increase IP	None
CO_OP_POP	Pop top value	Removes top stack item	None
CO_OP_LOAD_CONST	Load constant	Pushes constant onto stack	2-byte, constant index
CO_OP_LOAD_GLOBAL	Load global	Pushes global variable onto stack	2-byte, global index
CO_OP_LOAD_VARIABLE	Load local	Pushes local variable onto stack	2-byte, local index
CO_OP_STORE_GLOBAL	Store global	Stores top stack value in global	2-byte, global index
CO_OP_STORE_VARIABLE	Store local	Stores top stack value in local	2-byte, local index
CO_OP_BINARY_OP	Binary operation	Performs arithmetic/logical operation	1-byte, operation type
CO_OP_UNARY_OP	Unary operation	Performs unary operation	1-byte, operation type
CO_OP_CALL_FUNCTION	Call function	Calls a function	1-byte, argument count
CO_OP_RETURN	Return	Returns from function	None
CO_OP_JUMP_IF_FALSE	Conditional jump	Jumps if top stack value is false	2-byte, jump offset
CO_OP_JUMP_REL	Relative jump	Unconditional relative jump	2-byte, jump offset

6.2 Binary Operations

The VM supports the following binary operations, encoded as 1-byte operand values:

```
BIN_ASSIGN        ; Returns rhs object directly
BIN_ADD
```

```
BIN_SUB
BIN_MUL
BIN_DIV
BIN_MOD

BIN_EQ
BIN_NEQ
BIN_LT
BIN_LTE
BIN_GT
BIN_GTE

BIN_BIT_AND
BIN_BIT_OR
BIN_BIT_XOR
BIN_BIT_LSHIFT
BIN_BIT_RSHIFT

BIN_AND
BIN_OR
```

6.3 Unary Operations

The VM supports these unary operations:

```
UNARY_NEGATIVE          // Numeric negation (-)
UNARY_POSITIVE          // Numeric positive (+)
UNARY_LOGICAL_NOT       // Logical NOT (!, gayrı)

// Those unary operators get expanded to binary operations with one at bytecode
// generation so they do not directly live in bytecode
UNARY_PRE_INC           // (++variable)
UNARY_PRE_DEC           // (--variable)
UNARY_POST_INC          // (variable++)
UNARY_POST_DEC          // (variable--)
```

7. Memory Management

Manisa Engereği uses reference counting for garbage collecting. This approach is enough for most of the time but it is a bit problematic when objects have reference to each other (cyclic reference) a solution for that may be adding mark and sweep garbage collector beside reference counting but there is no time for this atm.

Every Manisa Engereği Object **(MEObject*)** has its own ob_refcount which are initialized 1 at start, we decrement / increment this reference count with macros (ME_DECREF, ME_INCREF) at everywhere we use those objects.

7.1 MEObject*

Meobjects are generic Manisa Engereği objects, they are the backbone of this interpreter. MEobjects allow us mimic a behaviour like polymorphism and inheritance in c this is both useful while creating internal functions at will come handy when classes will be added to language runtime because everything is already setup and things like operator overloading etc. are easy as just changing a function pointer with this setup.

Base MEobject structure:

```
typedef struct {
        size_t ob_refcount          // Reference count of object.
        METypeObject* ob_type       // Pointer to the type object.
} MEObject;
```

Every builtin object extends this definition

Every builtin type has its own type object defined in its respective file at vm/objects

```
struct METypeObject {
   const char* tp_name;
   METypeObject* tp_base;
   size_t tp_sizeof;
   fn_destructor tp_dealloc;
   fn_str tp_str;
   fn_bool tp_bool;
   fn_call tp_call;

   fn_nb_add tp_nb_add;
   fn_nb_sub tp_nb_sub;
   fn_nb_mul tp_nb_mul;
   fn_nb_div tp_nb_div;
   fn_nb_mod tp_nb_mod;
   fn_nb_bit_and tp_nb_bit_and;
   fn_nb_bit_or tp_nb_bit_or;
   fn_nb_bit_xor tp_nb_bit_xor;
   fn_nb_lshift tp_nb_lshift;
```

```
    fn_nb_rshift tp_nb_rshift;

    fn_unary_negative tp_unary_negative;
    fn_unary_positive tp_unary_positive;
    fn_unary_bit_not tp_unary_bit_not;


    fn_cmp tp_cmp;
};
```

(**Note:** Type objects are created statically at compile time if there is no inheritance (tp_base == NULL).)

For example floats have MEFloatObject* which is castable between MEObject* and itself and there is me_type_float type object for floats type dependent functions.

7.2 Stack Management

The stack-based architecture follows these principles:

- **Stack:** Stack is a dynamic array.
- **Stack Pointer (SP):** Points top of the stack.
- **Stack Operations:**
    - **`PUSH`:** Increments SP and stores a value
    - **`POP`:** Retrieves a value and decrements SP
    - **`TOP`:** Accesses the top value without removing it

7.3 Object Representation

All values in Manisa Engereği are represented as MEObject structures:

```
typedef struct MECodeObject {
    char* co_name;
    uint8_t* co_bytecode;
    size_t co_size;
    size_t co_capacity;
    HashMap* co_h_globals;
    HashMap* co_h_locals;
    MEObject** co_consts;
    MEObject** co_globals;
    MEObject** co_locals;
    uint8_t* co_lnotab;
    int in_function;
```

8. Execution Example

This section provides a comprehensive, step-by-step explanation of how the Manisa Engereği virtual machine executes a sample program, showing the complete lifecycle from source code to bytecode execution.

8.1 Sample Program

Consider this Manisa Engereği program that calculates a factorial:

```
marifet faktöriyel(n) {
    şayet (n == 0) {
        tebliğ 1;
    }

    tebliğ n * faktöriyel(n - 1);
}

çıktı("Sonuç: " + cümle(faktöriyel(5)));
```

8.2 Compilation to Bytecode

The compiler would generate the following bytecode structure:

Global Variables:
(Note: indexes from 0 to 10 are reserved for manisa engereği builtin io and typecast functions. You can find more info about them at vm/builtins**)**

```
0: çıktı
```

```
1: girdi
2: aç
3: kapat
4: oku
5: yaz
…
11:
```

Function Bytecode (faktöriyel):

```
Function: faktöriyel, nargs: 1
0000: LOAD_VARIABLE 0        ; Push function parameter (n) to stack.
0003: LOAD_CONST 2           ; Push number 0 to stack.
0006: BINARY_OP 6            ; Do binary “==” operation.
0008: JUMP_IF_FALSE 4        ; Jump 4 bytes forward if false.
0011: LOAD_CONST 3           ; Load 1 to the stack.
0014: RETURN                 ; Pop top and return.
0015: LOAD_VARIABLE 0        ; Push n to stack.
0018: LOAD_GLOBAL 11         ; Push faktöriyel function itself to stack.
0021: LOAD_VARIABLE 0        ; Push n to stack
0024: LOAD_CONST 4           ; Push 1 to stack
0027: BINARY_OP 2            ; Do binary “-” operation (n - 1)
0029: CALL_FUNCTION 1        ; Call function
0031: BINARY_OP 3            ; Do binary “*” operation (n * faktöriyel(n - 1))
0033: RETURN                 ; Pop and return.
```

Main Program Bytecode:

```
0000: LOAD_CONST 2           ; Load faktöriyel function as constant.
0003: STORE_GLOBAL 11        ; Store it in globals.
0006: LOAD_GLOBAL 0          ; Push çıktı function to the stack.
0009: LOAD_CONST 3           ; Push “Çıktı: “ string to the stack.
0012: LOAD_GLOBAL 9          ; Push builtin typecast function to the stack.
0015: LOAD_GLOBAL 11         ; Push faktöriyel function to the stack.
0018: LOAD_CONST 4           ; Push number 5 to the stack.
0021: CALL_FUNCTION 1        ; Call builtin cümle typecast function.
0023: CALL_FUNCTION 1        ; Call faktöriyel function.
0025: BINARY_OP 1            ; Add “Sonuç: “ and result of function.
0027: CALL_FUNCTION 1        ; Call “çıktı” to print string.
0029: POP                    ; Pop is added at the end of expr stmts always
                             ; because the result is unused.
```

10. Building Manisa Engereği and running a program written in Manisa Engereği

For linux users:

```
make && ./bin/me deneme.me
```

For windows users:

```
mkdir -p build
cd build
cmake ..
cmake --build .
.\bin\me.exe deneme.me
```

11. Possible Improvements

- Adding mark & sweep garbage collector for handling cycling references.
- Using arena allocator at frontend because there are too many needless malloc syscalls
- A hashmap like object, easy to implement but do not have enough time. This object is important for implementing classes
- More builtins.

12. More Code examples

(**Note:** Doc program uses different character for **“** changed most of it but there may be somes overlooked.)

Type Casting:

```
çıktı(ondalık("3")); # Prints 3.00

# Some operators like + handle type conversion automatically
çıktı(3 + 1.5); # 4.5

# Strings can be added together by default
çıktı("a" + "b"); # “ab”

# çıktı(3.14 + “b”) # Raises type mismatch error and halts vm
çıktı(cümle(3.14) + "hmm"); # “3.14hmm”
```

Taking Input From User:

```
# girdi function take variable args
# sabit yaş = girdi(); # is valid too
sabit hmm = girdi("Bir şeyler: ");
çıktı("Girdi: " + hmm);
```

Reading a File:

```
# Opens a file handle.
sabit dosya_handle = aç("test.txt", "r");
# Second argument means how many bytes to read from stream, -1 all .
sabit içerik = oku(dosya_handle, -1);
# Closes a file handle.
kapat(dosya_handle);

çıktı(içerik);
```