O Language for Newbies

Oytun Özdemir

Agust 2020

Contents

Part I

System Oriented Functional Programming Language

Chapter 1

What is System Oriented Functional Programming Language? (SOFPL)

Using your system's infrastructure, it is a programming language that allows you to develop programs on it. A syntax structure where you can control both the functional and the system there is. Code that you write in other programming languages and system functions to shorten purposes. Accordingly in other areas includes some features to make your work easier. In the example I gave below, I wrote it in C and O. you see a Hello World app.

1.1 Hello World for C and O language!

```
# include <stdio.h>
int main()
{
    printf("Hello, World!");
    return 0;
}
```

You have to type the Printf function into a main. Accordingly, we had to include the" stdio " library. This is because the printf function is in the "stdio" library.

```
show("Hello, World!")

println("Hello, World!")

"Hello, World!"
```

Here is simply show, println is a function. The Show is used only for the show. Println on line it is used to suppress text or objects. If the object if you write it alone,it will print out the last. We didn't include any library or anything because it was in the language because of the syntax structure, the

system is already in hibernation their function is to arrive. We have to see the result of the object. we can see by writing directly.

1.2 Installing

There are many methods to establish that language. The simplest is to download the olang executable file. Accordingly, docker is ready to be written in the Go programming language it has libraries.

1.2.1 Download or obtain Binary file

After you take this file and move it to your place, you can type *olang file* in Linux. you can run your ola files directly. or you can type in an olag to try and do direct experiments.

You can find the Binary file at this link: `http://gitlab.com/olanguage/olang`

1.2.2 Compile from source code

If you already have a go ecosystem, you can source it as follows: download the latest version in the GOPATH directory to both source codes you can look at it and compile your own tests. Depending on this, you can make your own improvements.

```
$ go get -v gitlab.com/olanguage/olang
```

You can find the source code address at the bottom of the link I will give.
`http://gitlab.com/olanguage/olang`

1.2.3 Work with that O Language on Docker

If you are saying You shouldn't take the risk. You can make improvements with that language on Docker. For example, if you have written a code but want to deploy it, the docker library is for you.

`http://hub.docker.com/repository/docker/olproject/olang`

To run the code directly on the Docker, use a you can follow the path.

```
docker run -t olproject/olang olang
```

Alternatively, type your own Dockerfile file on the Docker you can build.

```
FROM olproject/olang

RUN ["olang", "yourfile.ola"]
```

This way your own files and on the docker you can operate your work with olang.

Chapter 2

Dosyalarla çalışmak

If we go into a little more detail, that language is your ola extension they run or say your files. In this way more than one you can continue your operations by including the file.

```
def hello = fn(name){
    return ("Hello, "+name+"! Welcome to the O Language!")
}
def result = hello("Oytun")
show(result)
```

```
$ olang hello.ola
```

```
Hello Oytun! Welcome to the O Language!
```

You saved the file above and ran it as follows. You can practice on yourself by changing the contents. Let me explain to you exactly what we do now. First, we simply defined the function *hello*. We assigned it a *name* variable. All the last things we did we defined the variable *result* and the function *show* we showed it with. It's that simple.

2.1 File types we need to know

Simply specify the file types you need to know as tables. We'll find out the reasons by using them in the other sections.

Dosya Tipi	Uzantısı	İşlevi
O Language Program	.ola	O language main program or Program code
O Language Module	.olm	O Language Module File
O Language Process File	Opsfile	File running a direct O language process (no extension)
O Language Pacage File	olpfile.json	O Language Package File
O Language Library File	.oll	O Language Library File

Chapter 3

Working with modules

Modules actually take different functions from outside and include them in our program, it allows us to make different improvements. We can make libraries or additional modules.

Now an example module.ola write file.

```
literal hello(name){
    return ("Hello, "+name+"! Welcome to the O Language!")
}
```

A sample load to load the module. Let's create an ola file.

```
load "module.olm"
hello "Oytun";
```

Let's elaborate on the functions we do. we have *module* and *load*. One must have an "olm" extension because it is a module. The other is *load*. Our main program loads our module and its functions he'll transfer it to our main program. Where *literal* literal concept specifies that a function is literal. So previously if there is something about *hello*, overwrite it and replace it. As in the example, we have created the *hello* literal. So we changed the literature. Then we called this.

```
Hello Oytun! Welcome to the O Language!
```

As a result, our main program with the module is above us he ultimately gave the output involved.

Part II

Basic definitions and functions

Chapter 4

Definitions

In this Section more detailed functions and uses of functions we'll give you space. We will explain all the functions and explain how to use them.

4.1 How to define?

The constant is used to define. Defining variables as we will see in the example you'll be able to. To define a variable *def* usable.

```
def definition = object
```

4.2 Types

There are some species that we need when diagnosing variables. In this section we will define all species.

4.2.1 String and Integer

Text and numbers allow you to simply define text or Numbers. Various mathematical algorithms using text and numbers or you can do text operations. Text operations begin and end with ".

```
def string = "this is string"
```

In number operations, you must use a direct number.

```
def five = 5
```

As you can see, defining a variable is extremely simple. You can use these variables throughout the entire main application.

4.2.2 Logical Operations (Boolean)

This type of structure when looking at whether something is right or wrong we will use. It is extremely simple. The result is returned either *true* or *false*.

```
def right = true
```

```
def notright = false
```

This is how the definition is used.

4.2.3 Array and Hashes

Arrays are usually ordered structures. These ordered structures are like making a list. Each element of the list gets a number. Numbers are like a key to this hybrid structure. Using the key to open the door, it allows you to access the item behind the door.

```
def todo = [
    "Make Homework!",
    "Clean House"
]
println(yapilacaklar[0])
```

```
"Make Homework!"
```

That's how we made an introduction to the series. We have created a todo list and we showed him the first element of this list. Let's work on mixed structures. Let's say we have a key and various doors.

```
def keyring = {
    "key1": "room 2",
    "key2": "room 2",
    "key3": "garage",
    "key4": "door"
}
println(anahtarlik["key2"])
```

```
"room 2"
```

In the example, we arranged all the keys to which rooms to open. This way we can now see where *key2* will open. In this way you can make examples. The spelling is quite simple. You can write your elements directly into the arrays using "*[]*". In hash structures, you can define your elements as" *key : variable*". The only difference is that you have a key in one and the number (indexes) is used in the other.

4.2.4 Conditional Expressions (If/Else)

Functions that determine whether a job should be done or other work should be done, depending on a condition. *if* (if) and *else* (if not) are used in these conditions.

```
def wrong = true

if(wrong){
    println("That is really wrong!")
}else{
    println("That is not wrong!")
}
```

```
"That is really wrong!"
```

As you can see in the example, *error* is *true* because our variable is textitif the row in is returned. If *false*, the condition in *else* will work.

4.2.5 Functions

Now, the real issue is the functions that automate our lives without giving up. **What are these functions?** Functions are functions that simply run a function and variables that we write. Most of the time I do something over and over again we write a single function and we can use it many times by specifying its variables.

For example, let's say you go shopping and you don't have a to-do list. With an immediate example Let's write to-do list functions.

```
def list = []

def add = fn(todo){
    push(list, todo)
    println("Todo Added To List!")
}

def get = fn(num){
    return list[num]
}

add("tomatoes")
add("milk")

println(show(2))
```

```
"milk"
```

Here we used functions for example. We used three functions. These functions are *add*, *get*. All in *return* gives us back what is in it, or runs everything in it and returns the results. Here we have defined *materials*, with the function *add* into *list* with *push* using the function we had him send the attachments and returned the result to us with the number *get*.

4.2.6 Loops

Cycles have often saved lives. List the list or the same thing they are functions that repeat it rather than do it again. **do not write infinite loops when writing loops.** No one can stop something from going into an infinite loop. **because these loops are dangerous.** Lock the system. Let's write a cycle that doesn't go that far forever.

```
def i = 0;
loop(i>5){
    println("You cant do this 5 times!");
    def i = (i+1);
}
```

```
You cant do this 5 times!
You cant do this 5 times!
You cant do this 5 times!
You cant do this 5 times!
You cant do this 5 times!
You cant do this 5 times!
```

Here we simply defined the variable *i.* using *loop* , *i* we ran it until it was greater than 5 and each run we added another number to *i.* This *loop* function it is a form of work. *loop* continues to run until the specified ones are synchronized.

Let's look at this cycle differently. Let's be a little more restriction. I will use the *to be done* part of the loop that I wrote earlier. Let's make a list by seeing the items in this list and accessing their contents.

```
def list = [
    "Make Homework",
    "Clean House",
    "Eat Something",
]

for(list in number,value){
    def item = (itostr(number+1)+". "+(value));
    println(item)
}
```

```
1. Make Homework
2. Clean House
3. Eat Something
```

As you can see in the example, we've edited the entire to-do list. First of all, we had a *list* list here. This list *number* and *list* of each element in the loop *for* set to define and edit the variable *value*, we printed it out. The only different thing we use is the number with the *itostr* function we turned it into a string. The reason we do this is **number and text cannot be collected.**

4.2.7 Literals

The litareture structure allows us to modify the existing syntax in the system. The difference between literals and functions is that it means adding a function to the programming language. You can change or interfere with the functions in the system, the function written to you in the language or it allows you to expand methods. It is often used in modules.

Now I'm going to print out a math library for you as an example. The name of this file is *math.olm* will. The purpose of this library is simply to modulate the functions of simple calculators.

```
literal sum(x,y){
    return (x+y)
}

literal negative(x,y){
```

```
    return (x-y)
}

literal multiply(x,y){
    return (x*y)
}

literal divide(x,y){
    if(y == 0){
        return "infinity"
    }
    return (x/y)
}

literal fak(x){
    def i = 0;
    def result = 1;
    loop(i==x){
        def result = (result * x);
        def i = (i+1);
    }
    return result;
}
```

Our math library is ready to do its functions. Let's write a sample program using the litareture we have prepared. Let the program execute the factorial operation in *fak*. In addition, let's run the *sum* function.

```
load "math.olm"

println(fak(12))
println(sum(1,2))
```

```
8916100448256
3
```

As you can see, the result is *fak(12)* and *sum(1,2)* results we have already achieved. You can develop your own literature based on this sample library.

4.2.8 Scopes

The scope structure is often likened to class structures. Here is the scope structure it is used to collect and integrate many things in a single context. We just write *math*. Let's get the olm module covered and the module is in a scope must be found.

```
scope math{
    def sum = fn(x,y){
        return (x+y)
    }
```

```
    def negative = fn(x,y){
        return (x-y)
    }

    def multiply = fn(x,y){
        return (x*y)
    }

    def divide = fn(x,y){
        if(y == 0){
            return "infinity"
        }
        return (x/y)
    }

    def fak = fn(x){
        def i = 0;
        def result = 1;
        loop(i==x){
            def result = (result * x);
            def i = (i+1);
        }
        return result;
    }
}
```

We created a math scope in the example. We've added functions to it. Within this *math* scope, the group of functions is created and enclosed functions are assigned.

```
load "scope.olm"

def result = math->fak(10)
println(result)
```

```
10000000000
```

We called our module and the scope of *math* came up automatically. with the -¿ parameter, we sent a parameter to the*fak* function in the scope (we have accessed the textitfak function). We printed out the result.

4.2.9 Exceptions

Each programming language has a debug section. The Program starts a process first, and if something goes wrong in the process it will return to something else. If there is no error, the result will be output. This allows us to debug process management. Let's take a look at how it's used. The types you need to know here are *begin*, *except*, *recover*, and *final*. Let's explain the details of these with an example.

```
begin state {
```

```
    def wrong = true

    if(wrong){
        except state "That is wrong exception!"
    }else{
        final state{
            println("Finally it works!")
        }
    }

    recover state {
        println(error)
    }
}
```

```
That is wrong exception!
```

We have started a *state* definition, and we have started this definition we defined the variable *wrong*. Here it doesn't matter if you define the variable inside or outside. If the variable *wrong* is *true*, the *state* definition will generate an error process. apply the *recover* method to the *state* definition. Accordingly, from *except* define the message to *state* and use it as you wish. If a problem does not occur in the process *state* process sending *final* ensures smooth completion of the process.

Chapter 5

Input and Output

There are several methods to get information from the user from the terminal or to print something on the screen. *input* and *output* functions *show* and *println* functions on the O Language according to the new version it provides two additional options, depending on the different.

```
//input function:
def data = (input("Enter value: "))

//output function:
output("You entered: "+data)
```

```
Enter value: example
You entered: example
```

As the example shows, we have received a data entry from the *input* terminal, and we have it in the *data* variable . it even. Then we output it with the *output* function. These functions are technically *show* and *println* are similar functions.

Chapter 6

Math Operators

Mathematical operators have become an essential part of our lives. Apart from doing four basic operations with these operators, we do not use these operators in our functions. we became able to use it. These functions are extremely simple in That Language.

6.1 Basic Functions

In this section, we will see simple calculation functions. How to use them and use them in definitions We will learn.

6.1.1 Addition

The plus + sign is used for the addition.

```
def result = 2 + 2
inspect(result)
```

```
4
```

6.1.2 Extraction

The minus - sign is used for extraction.

```
def result = 2 - 2
inspect(result)
```

```
0
```

6.1.3 Divide

The slash / sign is used for the divide operation.

```
def result = 2 / 2
inspect(result)
```

```
1
```

6.1.4 Multiplication

Asterisk * is used for multiplication.

```
def result = 2 * 2
inspect(result)
```

```
4
```

For now, you can do math with these operations.

6.2 Yardımcı İşlevler

Here are a few simple functions that you can use for basic math operators in that language comes with. You can use it when you need them.

6.2.1 Sum

The sum function is similar to the sum function in mathematics. But here it is worth paying attention to by summing the two elements in the first two variables, it applies this to the set or function specified in the last variable.

```
def result = (math_sum(1,2, [1,2,3,4]))
inspect(result)
```

```
[3, 9, 18, 30]
```

As you can see, summing the numbers 1 and 2 directly to each element in the designated second sequence, he applied it to the cluster. We will explain this in more detail in the Mapping Functions section later.

6.2.2 Abs

If you give the number you want to get the absolute value of, as a result, you will get a number it will give its absolute value. The *abs* function is used here.

```
math_abs(-12)
```

```
12
```

It will simply take its absolute value and return the result.

Chapter 7

Geometry Functions

Some simple functions for you to perform geometric operations are available in this Language. This Help let's detail the functions with you.

7.0.1 Logarithm (Log)

You can use the *log* function to get the logarithm.

```
math_log(12)
```

```
2.4849066497880004
```

7.0.2 Cosine (Cos)

You can use the *cos* function to get cosines.

```
math_cos(12)
```

```
0.8438539587324921
```

7.0.3 Tangent (Tan)

To get a tangent, you can use the *tan* function.

```
math_tan(12)
```

```
-0.6358599286615807
```

Chapter 8

Comparison Functions

In this section, we will give the operators that you will use to compare the two expressions. You can make comparisons with these operators. In october, you can use the comparisons you make in internal functions and variables.

8.1 Equals

The equals operator, which allows you to compare two elements and get the right result when two elements are equal it will compare the two items and give the correct or incorrect result.

```
def falseResult = (1 == 2)
def trueResult = (1 == 1)

inspect(falseResult)
inspect(trueResult)
```

```
false
true
```

8.2 Is Not Equal

An equal operator that allows you to compare two elements and get the right result if the two elements are not equal it will compare the two items and give the correct or incorrect result.

```
def falseResult = (1 != 2)
def trueResult = (1 != 1)

inspect(falseResult)
inspect(trueResult)
```

```
true
false
```

8.3 Larger

It is the operator that provides the correct or incorrect result by looking at whether one element is larger than the other.

```
def brother = 10
def bigBrother = 20

inspect(brother > bigBrother)
inspect(bigBrother > brother)
```

```
false
true
```

8.4 Smaller

It is the operator that provides the correct or incorrect result by looking at whether one element is smaller than the other.

```
def brother = 10
def bigBrother = 20

inspect(brother < bigBrother)
inspect(bigBrother < brother)
```

```
true
false
```

8.5 Bigger than or Equal

An operator that checks the magnitude and equality of one element to another and returns a correct or incorrect result as a result.

```
def brother = 10
def olderSister = 20
def bigBrother = 20

inspect(olderSister >= bigBrother)
inspect(brother >= olderSister)
inspect(brother >= bigBrother)
```

```
true
false
false
```

8.6 Smaller than or Equal

An operator that checks the small and even volume of one element to another and returns a correct or incorrect result as a result.

```
def brother = 10
def olderSister = 20
def bigBrother = 20

inspect(olderSister <= bigBrother)
inspect(brother <= olderSister)
inspect(brother <= bigBrother)
```

```
true
true
true
```

Chapter 9

Mapping Functions

Mapping functions in a programming language by clustering every element in an element, the function is called the application function. Each element of the set is defined to a variable in the function. Array a single element is used. In mixed structures, two elements are used.

```
def numArray = [1,10, 30, 40, 50]

def result = (map(fn(x){
    return x*x
}, numArray))

inspect(result)

[1, 100, 900, 1600, 2500]
```

As you can see, we multiplied all the elements in the array with itself and returned them to the array.

Chapter 10

File Operations

Working with files is extremely easy. Let's start with the read and write functions. Every time you perform operations on files, you can change the file or change its contents doing work is the most necessary feature in every environment. Some programming languages this keeps things as long as possible. The functions you will see here are directly it's about the system.

10.1 File Read

To read a file, we simply use *read*. Read the file and set it to a variable we just need to assign it.

```
def test = read("deneme.txt")
```

We have read our file with the *read* function. Finally, we equated it to the trial variable. The resulting text will return.

10.2 File Write

To write a file, simply use *write*. Any stress or consequence prints directly to the file.

```
def content = "This is a test";
write("test.txt", content)
```

As in the example, we have printed our *content* text to our *test.txt* file.

10.3 File Delete

Dosya silmek için basit olarak *remove* kullanılır. Doğrudan dosyayı sistem tabanında siler.

```
remove("test.txt")
```

In the example, we deleted our *test.txt* file.

Chapter 11

Folder Operations

You will learn a few functions for performing simple operations on directories. In this section, you can work on directories. First of all, let's talk about the indexing function. With your existing user allows you to create an index.

```
mkdir("myfile")
```

we have created our directory with the *mkdir* function. Now let's delete this directory.

```
rmdir("myfile")
```

The directory and everything in it have been deleted. Now let's write a program with these file and indexing functions.

```
scope sys{
    def readFile = fn(file){
        return read(file)
    }

    def deleteFile = fn(file){
        return remove(file)
    }

    def writeFile = fn(file, content){
        return write(file, content)
    }

    def showEveryOne = fn(file){
        return chmod(file, 777)
    }

    def showToMe = fn(file){
        return chmod(file, 644)
    }
}
```

We have created scope for water functions sys; *readFile*, *writeFile*, *deleteFile*, *showEveryOne* and *showToMe*. These functions are to perform the functions written in the name. Only two functions call the *chmod* (change authorization) function. The *777* is authorized for everyone to see. The *644* permission is granted only for the user to see it at the moment. We will discuss the details of these parts in more detail in the *Permissions* section.

```
load "sys.olm"

sys->readFile("test.txt", "test text")
def deneme = sys->deleteFile("test.txt")
sys->showEveryOne("test.txt")
sys->deleteFile("test.txt")
```

We called our module and printed text to our *test.txt* file and read it. We called the function that says that everyone should see it. Then we deleted our file. All of them are functions included in the scope of *sys*.

Chapter 12

System Operations

We will touch on the changes that concern the operating system side. This part you can do only on the operating system it will consist of some important regulations or functions.

12.1 File Permissions

If you have knowledge of linux/unix, we assume that you have mastered the powers. If you are not a judge, we will talk about the powers in this section. Authorization on the system side is a set of functions that restrict or increase access to which file or folder you can access. For example, you have created a file that you do not want to be read. If you apply the necessary authorization to it after reading it, the user will not be able to access it. Only the file will be read by the program. We can program it so that the necessary actions are performed after reading it.

```
def myFile = "myFile.txt";
def readMyFile = read(myFile)
chmod(myFile, 000)
println(readMyFile)
chmod(myFile, 755)
```

As you can see in the example, I have read the file and applied the *000* (oct) authorization. The file can be read on the program side, but it will receive an authorization error when the user tries to open it. And then I gave him the authority back again. You can lock it while reading any file on the program side. The *chmod* function is sufficient for this. I'll tell you how to calculate these powers in a little while.

The parameters and their meanings can be found in the table below. To make a calculation, collect these powers. And write in threes, side by side.

For example, to grant write permission to a write group and user;

```
4+2+0 = 6
4+2+0 = 6
0+0+0 = 0

chmod(myFile, 660)
```

Parametre	Anlamı
u	User
g	Group
o	Other
a	Everyone
r	Read
w	Write
+	Give Permission
-	Take Permission
4	Read Permission (Numeric)
2	Write Permission (Numeric)
1	Execute Permission (Numeric)

You can use it in the form.

12.2 System Environment Operations

System variables allow you to access the system's variables. In addition, you can perform operations on these variables.

```
def path = sysenv("PATH")
```

For example, here we have checked the PATH variable in the system, and we have our own *path* variable we equalized.

12.3 System Commands and Processes

You need to run other commands on the system and use them to function we will summarize the functions in this section.

proc is used to start different types of operations in the system. For example if we want to list files in a directory in a different way, on linux/unix platforms *ls* we'll use your command.

```
def ls = (proc ls "ls" "-l")
println(ls)
```

.

```
..
forloop.ola
hello.c
hello.ola
load.ola
loop.ola
math.olm
mathtest.ola
merhaba.ola
modul.olm
proc.olm
scope.olm
scopetest.ola
sock.olm
sys.olm
systest.ola
todo.ola
```

You can define the result of the operation to a variable if you want.

12.4 Connections

We usually use ports to make connections through the system. After opening the port, we can communicate between the two computers by connecting to the port from another system. We'll see how to make these bilateral connections in a little while.

12.4.1 Socket Open and Listen

Bağlantı kurabilmek için öncelikle bir port açmak gereklidir. Bu işlemi yapabilmek için *sock* yapısını kullanmalıyız.

```
sock socket "tcp4" "9050" "0.0.0.0";
def messages = {
    "ping": "pong"
}
sock_listen(socket, messages);
```

As can be seen from the example, we have opened a socket for the *socket* variable. we have put the port on standby to respond to messages from the *tcp4 9050* port. If the *ping* message comes from the opposite side, the *pong* answer will be given.

12.4.2 Socket Connection

In order to connect to a port, you must first open the socket. After that, you can send messages to the port as raw using send.

```
sock socket "tcp4" "9050" "0.0.0.0"
sock_send(socket, "ping")
```

```
"pong"
```

As in the example, after opening the socket, we sent our message with *send*. Just you need to be careful the programs that make the connection and access the connection are two separate programs.

```
client: send -> ping -> listen (9050)
server: listen (9050) -> pong -> send
```

It can be exemplified in the way mentioned above.

Part III

Web Server and Database Operations

Chapter 13

Web Server

In this section, web-based studies are included in the system we will run a website with auxiliary functions. later We will simply link the website we have made to the database. General studies on routing and databases are based on that language you will learn how it can be written.

13.1 Routers

Router (Router), with any method they are structures consisting of queries to ports 80 or 443. Some header information to be able to query these routers it is sent and the machine on the opposite side responds to this header information it is desirable. If there is no such thing, various error codes are returned. The *webserver* function is used to write these routers in the O Language. This function is automatically called a web-based router by making various adjustments is generated. As a result, a web server runs when the code is executed.

```
def config = {
  "route path": {
    "type": "method :
                |GET
                |POST
                |PUT
                |DELETE
                |PATCH
                |OPTIONS
                |HOST
                |UPLOAD
                |STATIC
                |STREAM",
    "response": "response content",
    "input": "input content or data",
    "maxsize": "max file size (MB)",
    "headers": "header information",
    "folder": "folder path",
    "data": "data information",
    "host": "request accept only this host",
```

```
  }
}

webserver(port, config);
```

Specifying a setting hash array and port as specified above you can create a web server. We have specified all the details so that you can use it. Let's look at the ways of using methods.

13.1.1 GET

The Get method is the only method used to pull data.

```
"/get": {
  type: "GET",
  response: "Response Content"
  data: {}
}
```

The method of use is as follows above. Other entered values are invalid.

13.1.2 POST

The post method allows you to send data to the server. In October, in the data section parameters can be specified.

```
"/post": {
    type: "POST",
    response: "Hello {% parameter %}!"
    data: {
        parameter: "Oytun"
    }
}
```

The parameters can be rendered into the answer as follows;

```
{% parameter %}
```

In addition, what parameters will be october in the data section it should be given.

13.1.3 PUT

The put method is the method of throwing it inside. It works similarly with Post. With HTTP/2 it is an included feature.

```
"/put": {
  type: "PUT",
  response: "Hello {% name %}!"
  data: {
    name: "Oytun"
  }
}
```

```
{% parameter %}
```

In addition, what parameters will be october in the data section it should be given.

13.1.4 UPLOAD

Performs upload upload functions. Using the post method when files are sent to this address from the form or in any way automatically move to the folder specified in the *folder* parameter it is saved. In October, *input* is accepted from the form. If the file upload limit is to be specified, *maxsize* is specified in MB. If this is not specified, you may receive an error. That's why when writing your program you can keep the file upload limit at the highest value that can be useful to you.

```
"/upload": {
  type: "UPLOAD",
  input: "files",
  maxsize: "512",
  folder: "upload/"
}
```

13.1.5 DELETE

Delete is the deletion method that comes with HTTP/2. Delete functions in this method can be done.

```
"/delete": {
  type: "DELETE",
  response: "Deleted! {% data %}!"
  data: {
    name: ""
  }
}
```

13.1.6 STATIC

Static is used to list fixed files and return them it is a method. It lists only the files that are in the folder.

```
"/public/*filepath":{
    "type": "STATIC",
    "folder": "/public/"
}
```

The only thing to note here is the *filepath* address. Matched and related the file is recalled.

Now let's create a real web server with an example.

```
def config = {
    "/": {
```

```
        "type": "GET",
        "response": "Welcome to the O Language!"
    }
}
webserver("8080",  config);
```

```
$ curl localhost:8080
Welcome to the O Language!
```

We have opened a web server on the 8080 port in the example and GET it from the homepage we made sure that he answered the queries that came with his method. In this way, you can create your own using the settings and redirects you can create your web servers.

Chapter 14

Database Management

In this section, how to work with the database on that language is done. Simple as introduction you will receive. We will create models and create links on the database we'll do the study. Databases are used to connect to the database and to pull data. Databases it can usually be used to hold very large or small data. That language contains a database structure in itselfplays. If you want to connect with external databases you can install it. Or you can do file-based work on the internal database.

14.1 Model and Connections

In order to use the database, you must first establish a connection. A simple Model with sample codes on how you can connect we'll do the work.

```
def db = (database("type","connection address"))
```

Using the *Database* function, type the database and address to open the link we have connected it to the *db* variable.

```
def model = (model(db, "table", {
    "id": "int"
    ....
}))
```

Using the *Model* function, we have connected a table to the database we are connecting to to our model.

Now let's collect them all in one simple example and describe them in detail.

```
def connection = (database("internal","database"))

def user = (model(connection, "users", {
    "id": "int",
    "name": "text",
    "surname": "text",
    "username": "text",
    "password": "text",
}))
```

We have defined a *connection* and created our database we have set it to *internal*. We use our internal database. We have set this database to be stored in the *database* folder. We have created a user model and set it as the *user* model in the following sections we will use.

14.2 Table Create, Drop, Truncate (Migrate/-Drop/Truncate)

If the database supports *migrate* and *drop* operations, this section we will talk about the functions of deleting and creating tables.

```
//migrate function:
migrate(model)
//drop function:
drop(model)
```

The *migrate* function creates an object whose model we have already written it automatically creates a database and saves its history. if *drop* is this is the exact opposite. Destroys the specified database. Let's detail this with an example.

```
def connection = (database("internal","database"))

def user = (model(connection, "users", {
    "id": "int",
    "name": "text",
    "surname": "text",
    "username": "text",
    "password": "text",
}))

//drop function:
drop(user)

//migrate function:
migrate(user)
```

We have created a user model in our database and the model if there is, we first deleted it with the *drop* function. Then the *migrate* function we have recreated our table with.

As an alternative to these situations, the *truncate* function can be used. This function will delete the table instead of Delete the contents of the table.

```
truncate(model)
```

All data contained in the model will be deleted.

14.3 Create

The *create* function that we will use to create data in the database available. With this function, we can start saving data to the models we have created. In simple terms, the way it is used is as follows.

```
create(model, {...data...})
```

Veriler kısmında veriler yer alacağı için bunu bir örnekle gösterelim.

```
def connection = (database("internal","database"))

def user = (model(connection, "users", {
    "id": "int",
    "name": "text",
    "surname": "text",
    "username": "text",
    "password": "text",
}))

// create fonksiyonu:
create(user, {
    "id": "1",
    "name": "Oytun",
    "surname": "Ozdemir",
    "username": "oytun",
    "password": "1234"
})
```

We have created data on the connection we have already established and on our model. we created our data by sending data to our *user* model.

14.4 Update

We use *update* to update data in our database. We can use it to correct a mistake we made earlier or to update data. Simple usage is down.

```
update(model, {...data...})
```

The data that you will send from the Data section will be changed to the old one. New data is written instead. Let's detail this with an example.

```
def connection = (database("internal","database"))

def user = (model(connection, "users", {
    "id": "int",
    "name": "text",
    "surname": "text",
    "username": "text",
    "password": "text",
}))
```

```
// create function:
create(user, {
    "id": "1",
    "name": "Oytun",
    "surname": "Ozdemir",
    "username": "oytun",
    "password": "1234"
})

//update function:
update(user, {
    "id": 1,
},{
    "password": "gaiB3woo"
})
```

Using the *update* function at the bottom of the example i've updated my password. You can also experiment by updating your data.

14.5 Fetch

To pull data and convert relevant data into a variable according to a specific data from the database we have a *fetch* method for assigning. You can capture data using this method.

```
def results = (fetch(model, {...data...}))
```

Specifying what you need in the data part to be queried, you can specify the data that is relevant to it you can pull and synchronize to an array. Now let's detail this with an example.

```
def connection = (database("internal","database"))

def user = (model(connection, "users", {
    "id": "int",
    "name": "text",
    "surname": "text",
    "username": "text",
    "password": "text",
}))

// create function:
create(user, {
    "id": "1",
    "name": "Oytun",
    "surname": "Ozdemir",
    "username": "oytun",
    "password": "1234"
})
```

```
//update function:
update(user, {
    "id": 1,
},{
    "password": "gaiB3woo"
})

//fetch fonksiyonu:
def userFetch = (fetch(user, {"id": 1})
inspect(userFetch)

{
    "id": "1",
    "name": "Oytun",
    "surname": "Ozdemir",
    "username": "oytun",
    "password": "1234"
}
```

As you can see in the example, you can pull out the user with ID 1 i have synchronized it with the variable *userFetch* and with *inspect* we looked at the incoming data.

14.6 Query

If your database allows you to run a query, you can run a query.

```
def result = (query(model, "SQL query", {...parameters..}))
```

In order to run a query, you need to know SQL or a similar database. It allows you to run queries directly if you have SQL knowledge. In this case, Let's try.

```
def connection = (database("internal","database"))

def user = (model(connection, "users", {
    "id": "int",
    "name": "text",
    "surname": "text",
    "username": "text",
    "password": "text",
}))

// create function:
create(user, {
    "id": "1",
    "name": "Oytun",
    "surname": "Ozdemir",
    "username": "oytun",
```

```
    "password": "1234"
})

//update function:
update(user, {
    "id": 1,
},{
    "password": "gaiB3woo"
})

//fetch function:
def userFetch = (fetch(user, {"id": 1})
inspect(userFetch)

//query function:
def userList = (query(user,
        "SELECT * FROM users LIMIT ?",
        100))
```

The *query* function I wrote last will not work because my internal database does not support it. But it runs the SQL query on supported databases and adds the *UserList* variable to our he will send it. The purpose of this query is to limit and attract 100 users. If you have SQL knowledge, this way you can write queries. An important point to note here are the parameters leading to the SQL query it is added to the query by filtering and you will not have a SQL Injection problem. SQL Injection is a security problem. A security that allows you to run queries that are added to the parameter problem. In O language, these parameters are automatically checked and things entered as parameters are automatically cleaning is provided.

14.7 Delete

Deleting operations delete data from the database using queries or parameters provides. The *delete* function is used here.

```
delete(model, {...parameters...})
```

The data related to the parameters will be deleted using the model. Let's reinforce this with an example.

```
def connection = (database("internal","database"))

def user = (model(connection, "users", {
    "id": "int",
    "name": "text",
    "surname": "text",
    "username": "text",
    "password": "text",
}))
```

```
// create function:
create(user, {
    "id": "1",
    "name": "Oytun",
    "surname": "Ozdemir",
    "username": "oytun",
    "password": "1234"
})

//update function:
update(user, {
    "id": 1,
},{
    "password": "gaiB3woo"
})

//fetch function:
def userFetch = (fetch(user, {"id": 1})
inspect(userFetch)

//query function:
def userList = (query(user,
        "SELECT * FROM users LIMIT ?",
        100))

//delete function:
delete(user, {"id": 1})
```

As can be seen from the example, we deleted the user with ID 1.

In this section about the database, we have transferred the main business houses to you. You can complete this section by making examples yourself. In the next sections we will make examples on these databases.

14.8 Search

If your database supports it, the *search* function is available, which makes it easy for you to search. Let's summarize *search* function with an example.

```
def connection = (database("internal","database"))

def user = (model(connection, "users", {
    "id": "int",
    "name": "text",
    "surname": "text",
    "username": "text",
    "password": "text",
}))

// create function:
create(user, {
```

```
    "id": "1",
    "name": "Oytun",
    "surname": "Ozdemir",
    "username": "oytun",
    "password": "1234"
})

//search function:
def result = (search(user, {"name": "o"}))
```

Here it searches according to the parameters sent to the *search* function. The results return to us as a mixed setplays.

PS: **Not every database may support this method. In order to support it, the most indexing feature must be present in the database.**

Part IV

Helper Functions

Chapter 15

What is helper functions/internal functions?

Helper functions should usually be in a programming language it contains features that help with the functions performed. In this section, the details of these functions we will talk about.

15.1 File Render

Create a new variable by sending some parameters to the file contents in that Language file rendering operations are available that help.

```
def result = (renderf("file.txt", {...data...}))
```

The result is added to our variable.by sending data to the file.txt content, a new we have created text. Variables can be defined as;

```
{% parameter %}
```

In this way, the data coming from outside is replaced with *parameter*. The new the text can be used as desired.

15.2 XML Helpers

XML files are not only in the XML content that we pull from the url *xmlf* functions that we can use when working with normal XML it contains. When working with XML (X Markup Language) files, you can use the *read* function and retrieve its contents it was developed in order not to write too much code on it. To send an XML parcel, use *xmlf* its function will be useful to you.

```
def result = (xmlf("file.xml"))
```

Using the *xmlf* function on the result variable file.i pulled my xml file and we defined it.

If you want to convert text, not a file, you can use the *xmlp* function.

```
def result = (xmlp("...xml content..."))
```

Using the *xmlp* function on the result variable we have defined the content written in xml.

15.3 JSON Helpers (JSON encode/decode)

It is quite easy to work with JSON files. For this, we have a few simple functions available. These are the *jsonp* and *jsonf* functions.

```
//jsonp function:
def result = (jsonp("json content"))
//jsonf function:
def result = (jsonf("file.json"))
```

jsonp function displays the contents of the incoming JSON as text converts it to a mixed array and, as a result, returns an object.

the *jsonf* function automatically retrieves a JSOn file by converts it to a hash array and, as a result, returns an object.

Using these two functions, you can create JSON content that comes from a file or as text You can use it on that Language.

15.4 Color Helper (Colorize)

It is an auxiliary function developed for receiving color output from the terminal. If your terminal supports colors you can get color output.

```
colorize("red", "Hello Red!")
colorize("blue", "Hello Blue!")
colorize("green", "Hello Green!")
colorize("yellow", "Hello Yellow!")
colorize("grey", "Hello Grey!")
colorize("black", "Hello Black!")
colorize("bold", "Hello Bold Color!")
colorize("magenta", "Hello Magenta!")
colorize("white", "Hello White!")
colorize("red", "Hello Red!", true)
colorize("blue", "Hello Blue!", true)
colorize("green", "Hello Green!", true)
colorize("yellow", "Hello Yellow!", true)
colorize("grey", "Hello Grey!", true)
colorize("black", "Hello Black!", true)
colorize("bold", "Hello Bold!", true)
colorize("magenta", "Hello Magenta!", true)
colorize("white", "Hello White!", true);
```

You can use the colors directly on your terminal.

15.5 Regular Expressions

Regular expressions make it easy to process complex texts today it can be characterized as groups of parameters. How to set these parameters on that language let's see it processed.

```
def rex = "[A-Z][a-z]"
def result = regexp_check(rex, "Hello!")
true
```

The *check* method verifies whether a regular expression given in a text exists. As a result, it provides us with a correct or incorrect return.

```
def rex = "he"
def result = regexp_find(rex, "Hello!")
["he"]
```

The *find* function finds what is in it for us and returns it as an array. In this way, we can perform extraction and verification on the text using regular expressions.

15.6 Encryption Helpers (Encrypt/Decrypt)

They are an auxiliary function used to encrypt text or encrypt content. These encryption functions encrypt the text and help hide it.

```
// example encrypt:
def encText = (encrypt("KEY", "text"))

// example decrypt:
def decText = (decrypt("KEY", encText))
```

As in the example, we encrypted the text using the key to our *encText* variable. The *encText* variable cannot be used or output here. We can save it by writing it to a file. It may be necessary to solve it because its contents are not displayed. Using the key in the second *decText* function to decode we can decode the text and get to the text. As you can see, it is quite easy to use.

15.7 Replace Helper

When working on texts, the replace function is used to find and change the text. This function finds and modifies the text specified in the text.

```
def text = "Hello content!"
def text = (replace(text, "content", "O Language"))
output("%s",text)

"Merhaba O Language!"
```

As you can see, we replaced the word "content" with "O Language" using *replace*. You have your own tests you can do it.

Part V

Dangerous Waters (Biohazard)

Chapter 16

Why?

The purpose of this part is to be critical and finally to explain the purpose of these critical processes in the system the fact that it is running directly on your side and the actions you will take in this part may damage your system. If you need very low-level programming functions, the operations here are on the system side it can make you do physical work.

PS: **This works with very low-level functions and low-level programming knowledge is also required.**

Chapter 17

Use Text as Code (Eval)

This can be used in very critical situations. Enter a code *eval* as text you can make a text run directly using the function. The harmful thing about this function is that if there is an encrypted content, you can use it by *decrypt* you can execute. **In this case, calling unknown codes can damage the system.**

We will give a dangerous example. This example will only be useful for you to understand the incident in more detail.

```
def code = "def x = 1";
(eval(code))
inspect(x)
```

```
1
```

As you can see in the example, our *code* variable must contain a code. we ran this with *eval*. The code in it worked automatically, and the variable x he was identified.

Now we will give you a more dangerous example.

```
// key.txt:
111-222-444
```

```
// source.ola:
def x = 1
```

```
// encrypt.ola:
def key = (read("key.txt"))
def source = (read("source.ola"))
def file = (encrypt(key, source))
write("encrypted.ola", file)
```

```
// decrypt.ola:
def key = (read("key.txt"))
def file = (read("encrypted.ola"))
def result = (decrypt(key, file))
eval(result)
inspect(x)
```

```
1
```

Now let's explain this example. We have created a *key* file to hold the key. This file will carry the key that will open the file. our *source* file will contain the codes to be encrypted. *encrypt* will be our encryption program. our program *decrypt* will decrypt and execute the code. When decrypting *key.txt* file you will use. Since the *encrypted* file cannot be read in any way, the user will not see the source of the code. This way the file will work encrypted.

In this section, the danger is exactly in the *encrypted* file. Because this code is encrypted, it is a dangerous it may contain source code. This may cause damage or damage to the computer. Or it can be decoded and added to the code by someone else and encrypted again. This is a dangerous process.

Chapter 18

Machine Code Execute (Assembly)

Running Assembly operators directly is an extremely risky situation. A virtual processor directly on the machine these codes are operated on it, and the machine directly physically operates these codes. **Execute the codes in this section it can provide extremely dangerous consequences. If you don't have Assembly knowledge, you should first know what these codes are running.**

Here we will only show you how it works. Therefore, instead of giving examples, you should test these codes on your own side. To use these operators, the *asm* function is used.

```
asm(operationCode, parameters, parameters...)
```

Let's test our code by making an example hello world.

```
def hello = "Hello!";
def len = len(hello)

asm("DB", "0xa", hello)
asm("EQU", len)
asm("MOV", "edx", len)
asm("MOV", "ecx", msg)
asm("MOV", "ebx", 1)
asm("MOV", "eax", 4)
asm("INT", "0x80")
asm("MOV", "eax", 1)
asm("INT", "0x80")

Hello!
```

By using Assembly DB, you can access the *0xa* address in memory with the command "Hello!" we wrote down our variable. Calculate its size and start at *0xa* in memory and wait until it reaches this value that this message will be located we have indicated. We moved all these values to the *edx* and *ecx*

addresses. we have sent 1 to the *ebx* block. This means output the system. we sent 4 to *eax*. This means that this data block is added to the system, that is it means that you write data with the value *0xa* in memory and the size *eq*. we have sent *0x80* to the *int* block. It has the function of calling from the system. we sent 1 to *eax*. and we ran the callback function again. It means that you call the variable sent to the system and send an output signal to the system.

In this way, we have started machine operations. We processed the machine code to the low-level processor to return the result.

Chapter 19

Execute Programs (Open/Run/Exec)

Running commands may not actually be dangerous. But sometimes the functions here can be used to run malicious software. **These commands can damage our system.** We still want you to find out. Therefore, we will summarize their functions in this section.

19.1 Open Programs (Open)

The open command allows you to open and run programs directly on the system. Using this command you can open an application on the system.

```
open("program", ...parameters...)
```

You can open the application using the program and parameters.

```
open("firefox")
```

As in the example, we ran firefox. If your system is suitable for running the interface firefox they will work.

19.2 Run Applications (Run)

The command execution function allows you to execute different commands. **You may not be able to control running commands if you run commands without creating a process.**

```
run("program", ...parameters...)
```

As an example, let's run a program and synchronize its results with a variable.

```
def result = (run("ls", "-la"))
```

We run the listing command and synchronize it with the variable *result*.

19.3 Execute commands on OS shell (Exec [SH/CMD])

If you know which system you are working on, start the appropriate system startup functions and; you can synchronize the results to a variable.

sh works only on linux and unix-type systems:

```
def result = (sh_exec("ls", "-la"))
```

cmd works only on windows-type systems:

```
def result = (cmd_exec("dir", "/w"))
```

In this way, you can run platform-specific commands that you are working with. In two *result* variables executes the listing commands. In this way, you can run your own commands.

PS: **Running a program or command without opening a transaction in this way is like performing an operation in an extremely uncontrolled way. You can damage your system. It will be more convenient for you to run a command by opening a process as we mentioned in 7.3.**

Part VI

Example Real World Applications

Chapter 20

Web Framework Development

Web Frameworks provide various auxiliary functions when creating or running a website they are structures that contain. Access to a website so that people can view the codes written as a whole they transform it.

To make a web roof, first determine how to make a folder structure. Let's start with a simple folder structure and determine the other structure.

```
boot.ola
config
controllers
database
library.olp
proc.ops
public
templates
```

Boot the program.we will run it with the ola file. By reading the config files, By including controllers files; It will perform rendering operations via Templates. It will display the pages to the person browsing the site in accordance with the redirects on the Config. Libraries downloaded from outside are kept independently under vendor. If it is in the Public folder, it is open to everyone the files will be listed. Now let's look at the source codes of our Application. You can download these source codes we will share the address at the end of the study.

```
// boot.ola:
# Controllers

load "vendor/ondle.ola"
load "controllers/index.olm"

# Configuration Definitions
load "config/config.ola"
```

```
println("Server running on 0.0.0.0:80.")
webserver("80", config)
```

Our boot file is ondle under vendor.the ola file is the controller/index.the olm module, config/config.it starts a web server on port 80 by including ola setting files.

```
// config/config.ola:
def config = {
  "/": {
    "type": "GET",
    "response": (main())
  }
};
def db = (database("internal","database"));
```

There are routing and database variables in our Config file.

```
// controllers/index.olm:
def main = fn(){
  return (renderf("templates/index.html",
    {
      "title": "Olang Web Framework | Index",
      "body": "Welcome to the Olang Web Framework."
    }
  ));
};
```

The index controller module templates/index.which writes the title and body variables to the html file, makes a main definition.

```
// library.olp
{
        "package": "olf",
        "publisher": {
                "name": "Oytun",
                "email": "info@oytun.org",
                "github": ""
        },
        "sources": "src",
        "main": "boot.ola",
        "requires": [
                {
                        "name": "olang-html",
                        "src": "https://gitlab.com/olanguage/libraries/olang-html.git"
                }
        ],
        "compiled": "",
        "install": null,
        "jobs": [
                {
```

```
                        "name": "start-web",
                        "command": "olang process --start true"
                },
                {
                        "name": "stop-web",
                        "command": "olang process --stop true"
                }
        ],
        "type": "project"
}
```

A library with libraries and requirements.the olp file,

```
$ olang install
[!] INFO: "vendor" path not found. Path creating...[ready]
[>] Fetching    [olang-html][done]
[>] Compiling   [vendor/olang-html/src/main.ola]
[>] Compiling   [/home/ted/workspace/olf1/vendor/olang-html/src/component.ola]
[>] Compiling   [/home/ted/workspace/olf1/vendor/olang-html/src/html.ola]
[>] Compiling   [/home/ted/workspace/olf1/vendor/olang-html/src/form.ola][done]
[>] Writing Autoload... [vendor/ondle.ola][done]
[X] Fetch Completed.
```

it automatically loads libraries under vendor when the installation command is run.

```
// proc.ops
[olang]
boot: boot.ola
```

```
$ olang process --start true
2020/08/15 18:35:06 Running Proc: 5874
2020/08/15 18:35:06 open nohup.out: no such file or directory
```

Proc.our ops file specifies which process will be run in the background as a process with which name. When we call the Process command, it will automatically run the functions connected to this file and our website will be available it will open.