Recursion

This lesson will get you acquainted with recursion in Rust.

We'll cover the following

What Is Recursion?

Recursion is a method of function calling in which a function calls itself during execution.

There are problems which are naturally recursively defined. For instance, the factorial of a number n is defined as n times the factorial of $n - 1$.

```
factorial(n) = n * factorial(n-1)
```

Parts of Recursion

In terms of programming, a recursive function must comprise two parts:

- **Base case**

  A recursive function must contain a base case. This is a condition for the termination of execution.

- **Recursive case**

  The function keeps calling itself again and again until the base case is reached.

Example

The following example computes the factorial of a number using recursion:

> **Note:** A factorial is defined only for non-negative integer numbers.

```
// main function
fn main(){
    // call the function
    let n = 4;
    let fact = factorial(n);
    // print the factorial
    println!("factorial({}): {}", n, fact);
}
// define the factorial function
fn factorial(n: i64) -> i64 {
    if n == 0 { // base case
        1
    }
    else {
        n * factorial(n-1) // recursive case
    }
}
```

Explanation

- `main` **function**

  The `main` function is defined from **line 2 to line** 7.

  - On **line 4**, a call is made to function `factorial` with an argument passed to the function and the return value is saved in the variable `fact`.

  - On **line 6**, the value of the variable `fact` is printed, i.e., the factorial of the number being passed as an argument.

- `factorial` **function**

  The `factorial` function is defined from **line 9 to line 16**.

  - **function definition**

    - The function takes a parameter `n` of type `i64`.

  - **function body**

    The recursive function is made up of two parts.

    - **base case**

On **line 10**, the base case is defined. Since the value of `n` is decremented in every recursive function call, the function terminates when the value of `n` becomes equal to `0` on successive recursive calls.

- **recursive case**

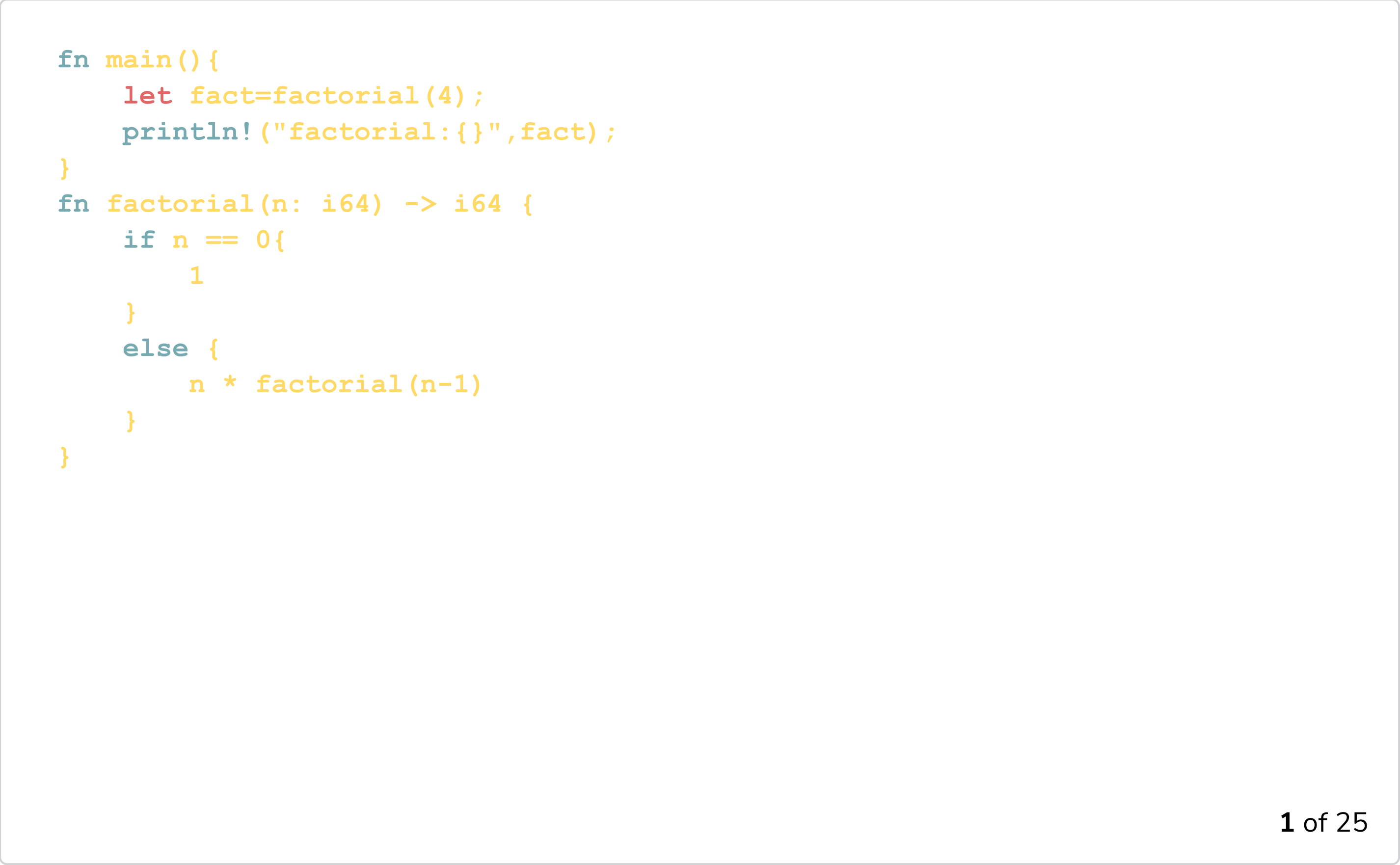
  On **line 14**, the recursive case is defined. The value `n` gets multiplied with `factorial(n-1)` and gets pushed on the memory stack. Since the value of `n` is decremented in every function call, the function keeps on calling itself repeatedly until the base case is reached. As soon as the base case is reached, the $factorial(0)$ is calculated and the value is used in the immediate expression in the memory stack. The $factorial(1)$ is calculated from $1 * factorial(0)$. $factorial(2)$ is calculated from $2 * factorial(1)$. This process $n * factorial(n-1)$ continues until the last value is freed from the memory stack.

The following illustration shows how $factorial(4)$ is computed:

- $factorial(4) = 4 * factorial(3)$ => This memory frame gets pushed on top of stack
- $factorial(3) = 3 * factorial(2)$ => This memory frame gets pushed on top of stack
- $factorial(2) = 2 * factorial(1)$ => This memory frame gets pushed on top of stack
- $factorial(1) = 1 * factorial(0)$ => This memory frame gets pushed on top of stack
- $factorial(0) = 1$ => base case reached

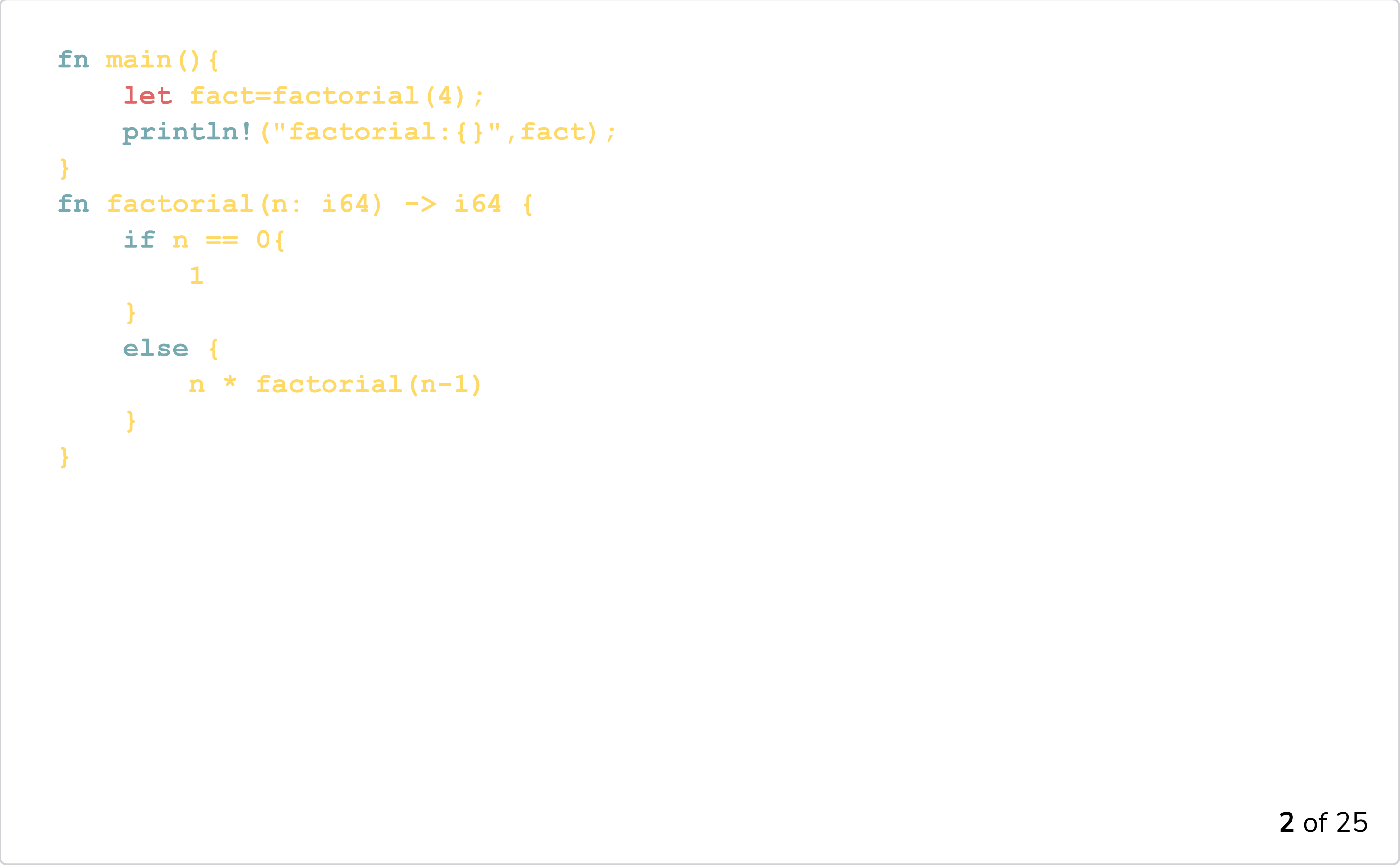
  - $factorial(0) = 1$
  - $factorial(1) = 1 * factorial(0) = 1 * 1$ => This memory frame gets freed
  - $factorial(2) = 2 * factorial(1) = 2 * 1$ => This memory frame gets freed
  - $factorial(3) = 3 * factorial(2) = 3 * 2$ => This memory frame gets freed
  - $factorial(4) = 4 * factorial(3) = 4 * 6$ => This memory frame gets popped off, the allocated memory is released and eventually the value is returned.

The following illustration explains the above code:



```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

1 of 25

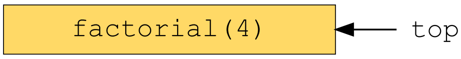


```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

2 of 25

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

4

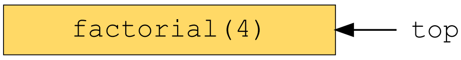


factorial(4)

top

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{  false
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

4

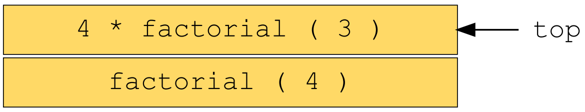


factorial(4)

top

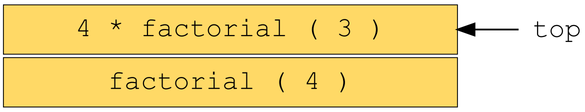
```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

4



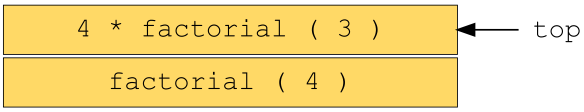
```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

4



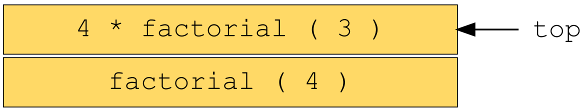
```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

3



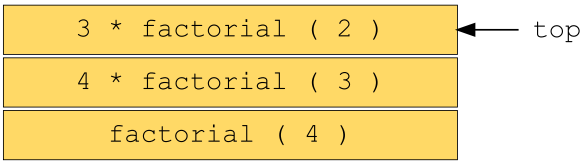
```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{  false
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

3



```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

3



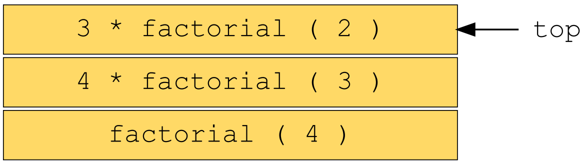
3 * factorial (2) ← top

4 * factorial (3)

factorial (4)

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

2



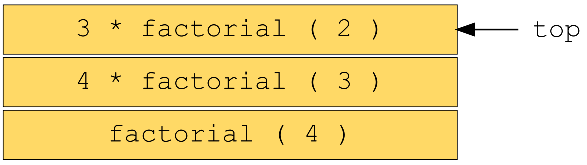
3 * factorial (2) ← top

4 * factorial (3)

factorial (4)

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{ false
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

2



3 * factorial (2)

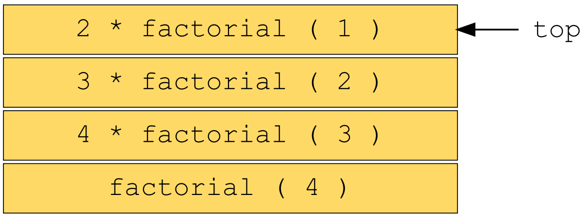
4 * factorial (3)

factorial (4)

top

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{ false
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

2



2 * factorial (1)

3 * factorial (2)

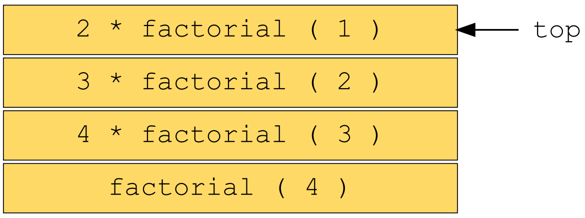
4 * factorial (3)

factorial (4)

top

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

1



2 * factorial (1)

3 * factorial (2)

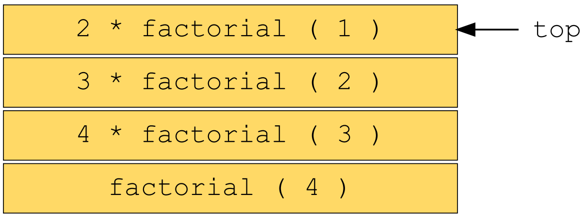
4 * factorial (3)

factorial (4)

top

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{  false
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

1



2 * factorial (1)

3 * factorial (2)

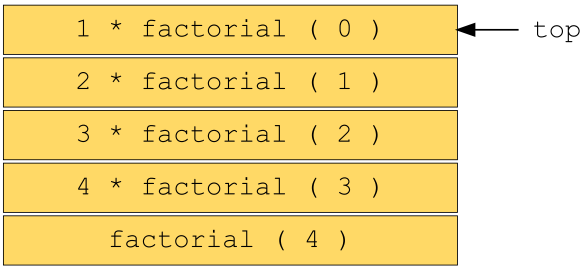
4 * factorial (3)

factorial (4)

top

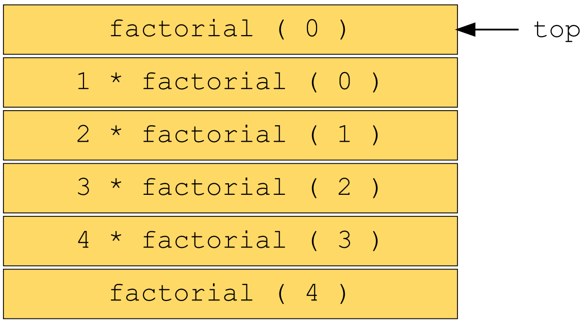
```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{  false
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

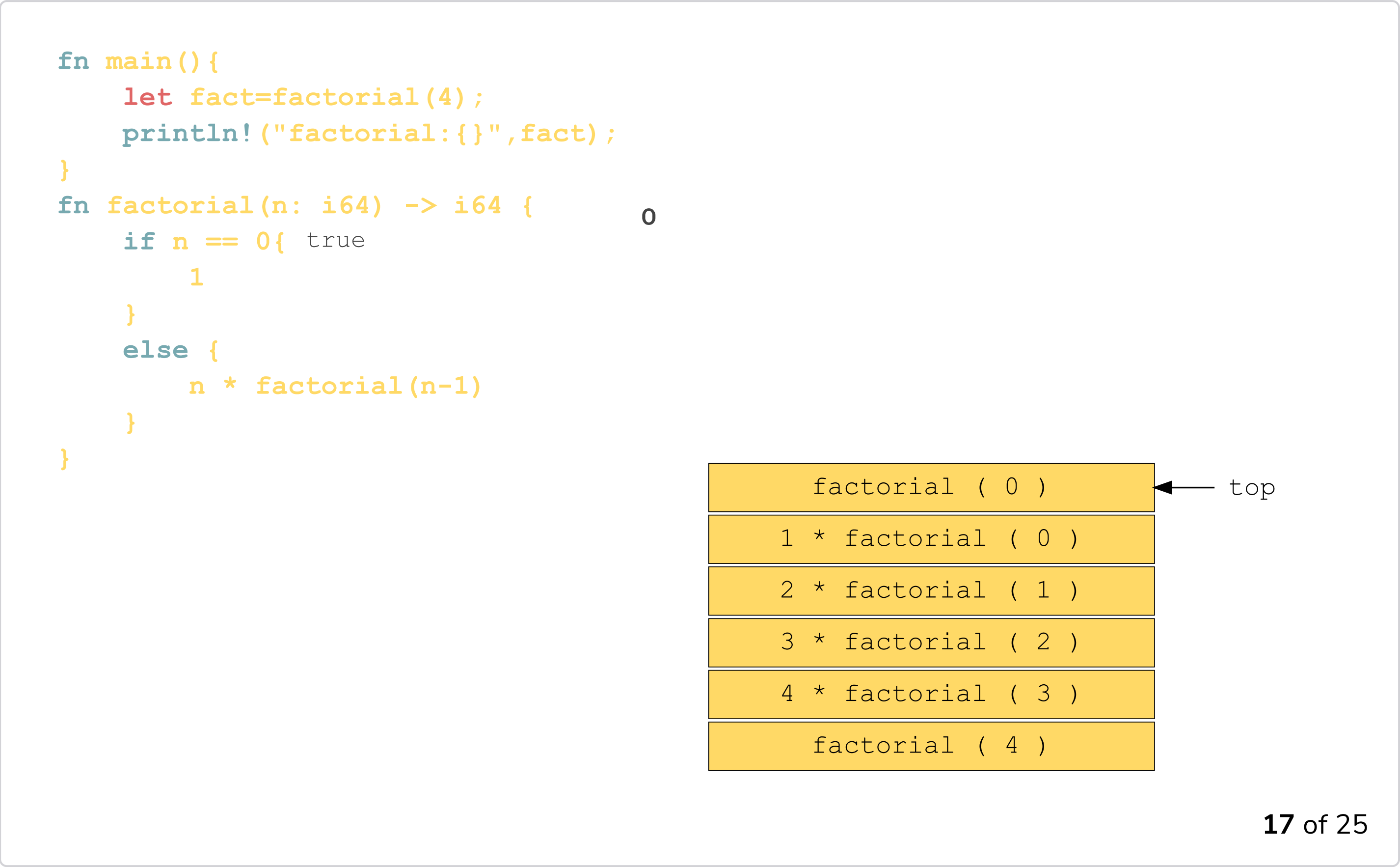
1



```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

0





```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{ true
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

0

factorial (0) ← top

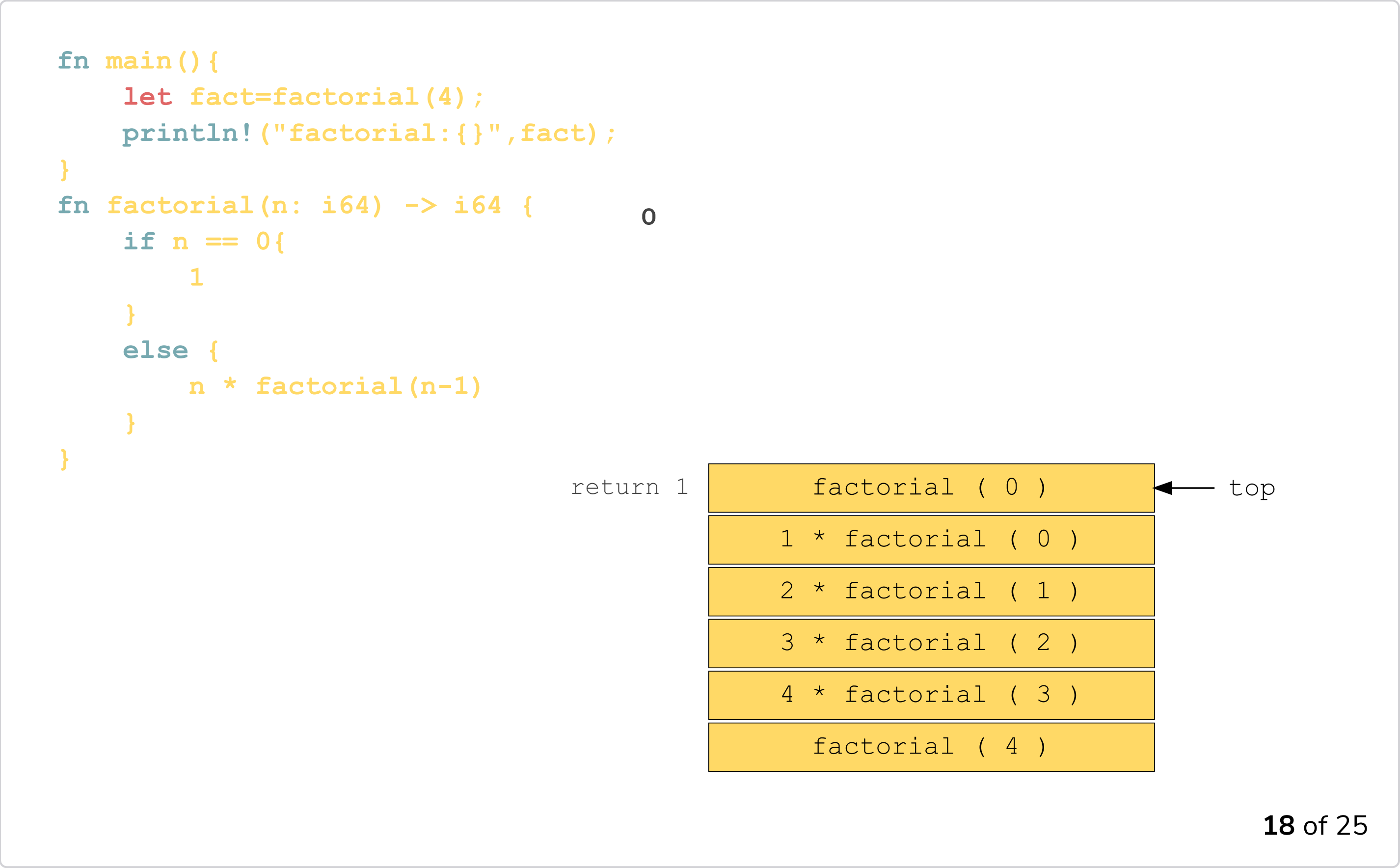
1 * factorial (0)

2 * factorial (1)

3 * factorial (2)

4 * factorial (3)

factorial (4)



```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

0

return 1

factorial (0) ← top

1 * factorial (0)

2 * factorial (1)

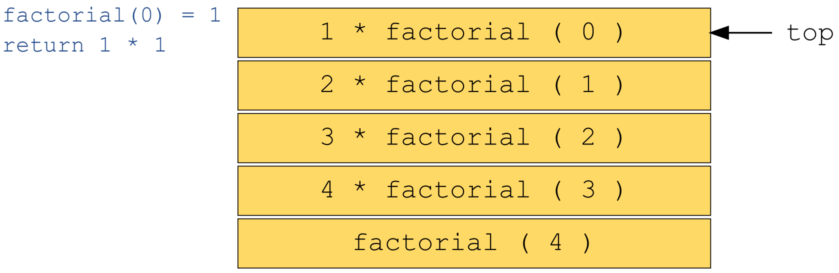
3 * factorial (2)

4 * factorial (3)

factorial (4)

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

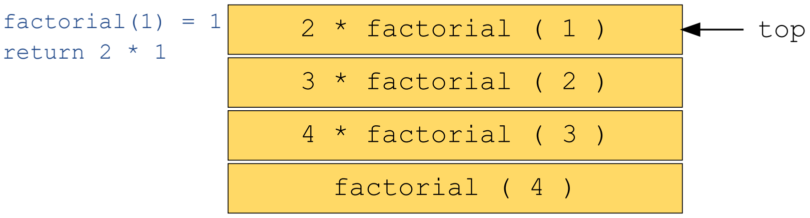
0



factorial(0) = 1
return 1 * 1

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

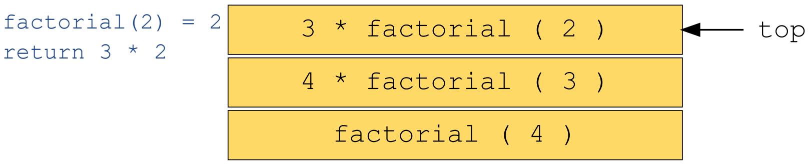
0



factorial(1) = 1
return 2 * 1

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

0



factorial(2) = 2
return 3 * 2

3 * factorial (2)

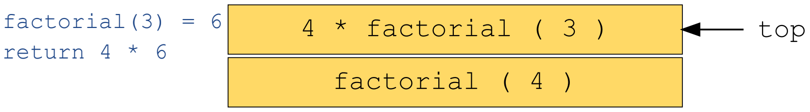
4 * factorial (3)

factorial (4)

top

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```

0



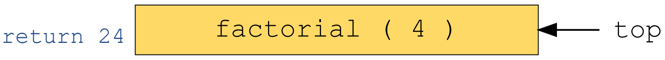
factorial(3) = 6
return 4 * 6

4 * factorial (3)

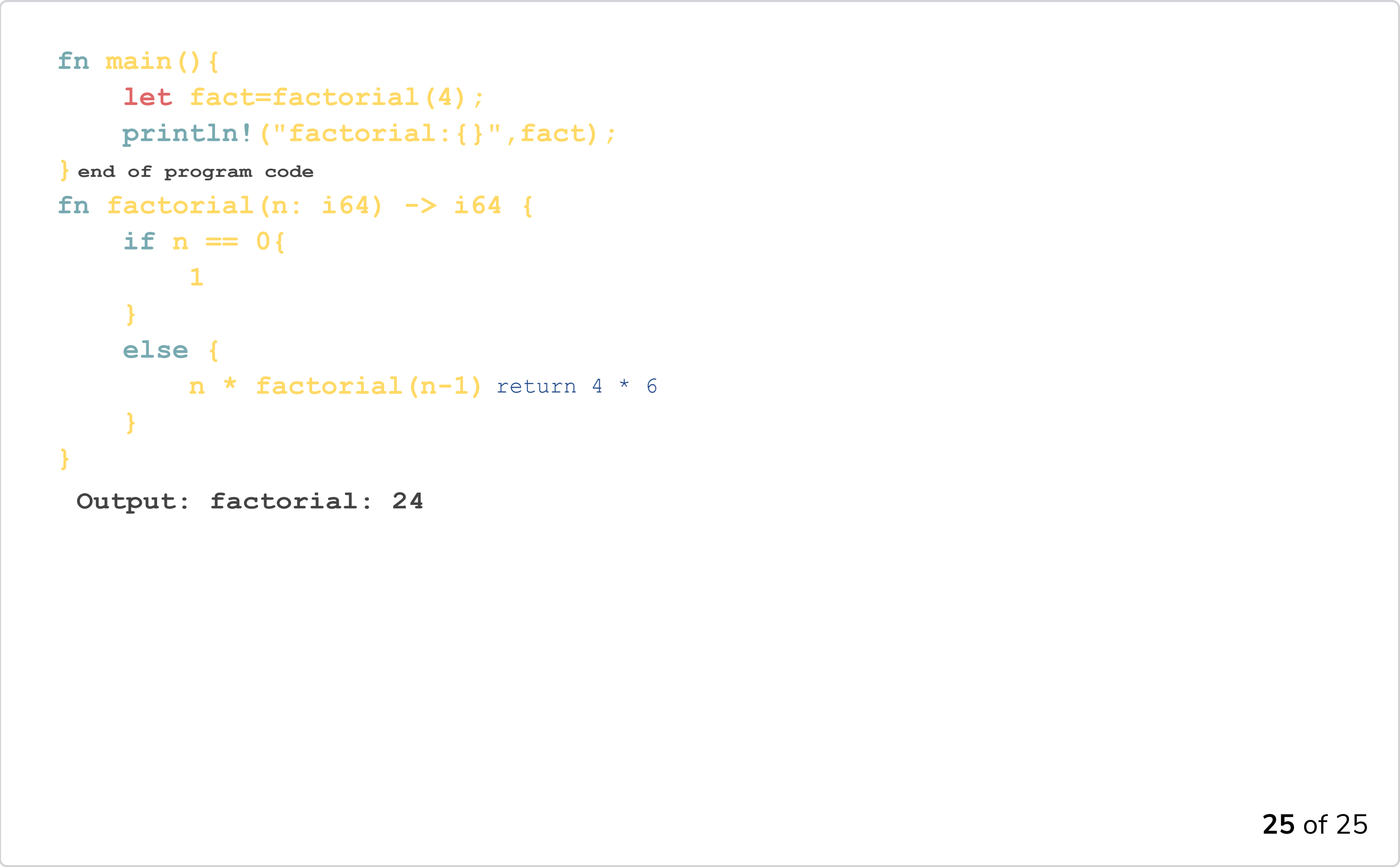
factorial (4)

top

```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {      0
    if n == 0{
        1
    }
    else {
        n * factorial(n-1)
    }
}
```



```
fn main(){
    let fact=factorial(4);          24
    println!("factorial:{}",fact);
}
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1) return 4 * 6
    }
}
 Output: factorial: 24
```



```
fn main(){
    let fact=factorial(4);
    println!("factorial:{}",fact);
} end of program code
fn factorial(n: i64) -> i64 {
    if n == 0{
        1
    }
    else {
        n * factorial(n-1) return 4 * 6
    }
}
 Output: factorial: 24
```

Now that you have learned about recursive functions, solve a challenge before moving on to the next chapter.