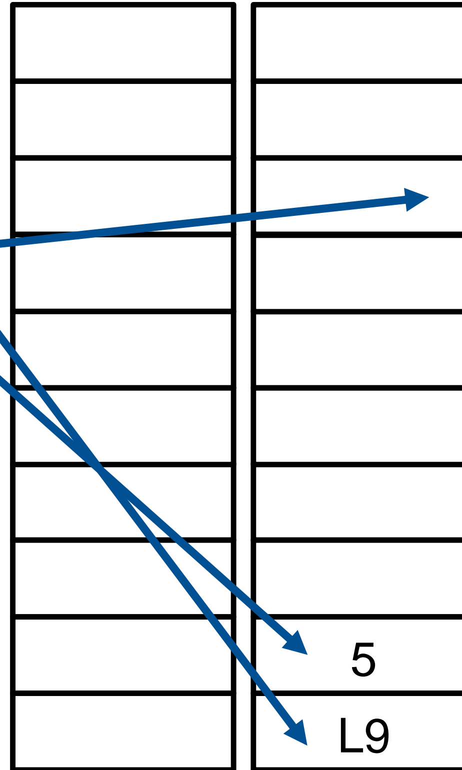
Chapter 2: The Recursion Principle and a Few Basic Algos

Execution of Recursion with a Stack

Trace of Execution:

When facing line 9, the computer will
- Put Line 9 onto the stack
- Put 5 onto the stack
- Call (jump) to Line 1




```
1  function ret = factorial(n)
2          if n == 0
3                  ret = 1
4          else
5      ret = n * factorial(n-1)
6          end
7  end
8
9  factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Now facing line 1,

- Take x from the stack (x=5)
- Run program until Line 5
- Put L5 onto the stack and call
- Put 5-1 = 4 onto the stack
- Call (jump) to Line 1

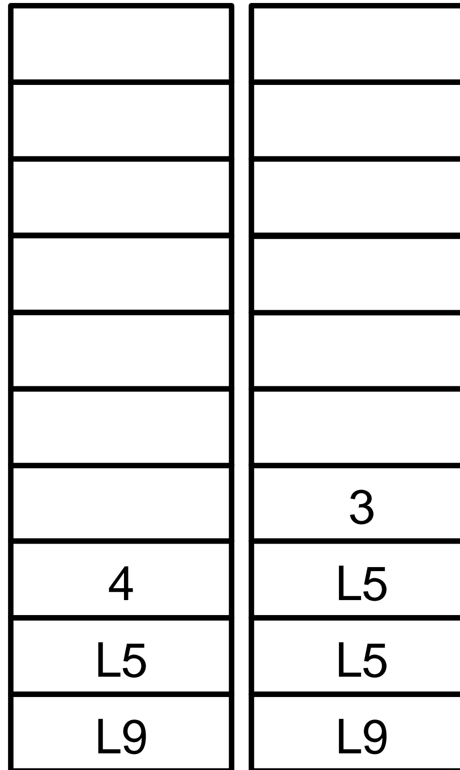
```
1  function ret = factorial(n
2          if n == 0
3                  ret = 1
4          else
5      ret = n * factorial(n-1)
6          end
7  end
8
9  factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Now facing line 1,

- Take n from the stack (n=4)
- Run program until Line 5
- Put L5 onto the stack and call
- Put 4-1 = 3 onto the stack
- Call (jump) to Line 1



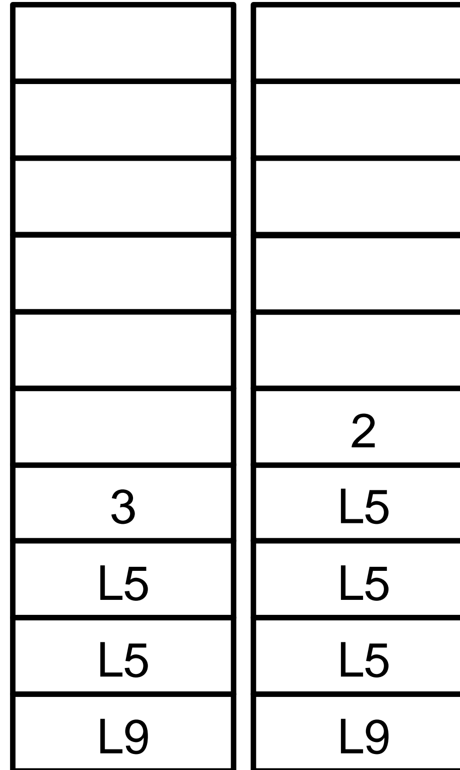

```
1  function ret = factorial(n)
2      if n == 0
3          ret = 1
4      else
5    ret = n * factorial(n-1)
6      end
7  end
8
9  factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Now facing line 1,

- Similar as before



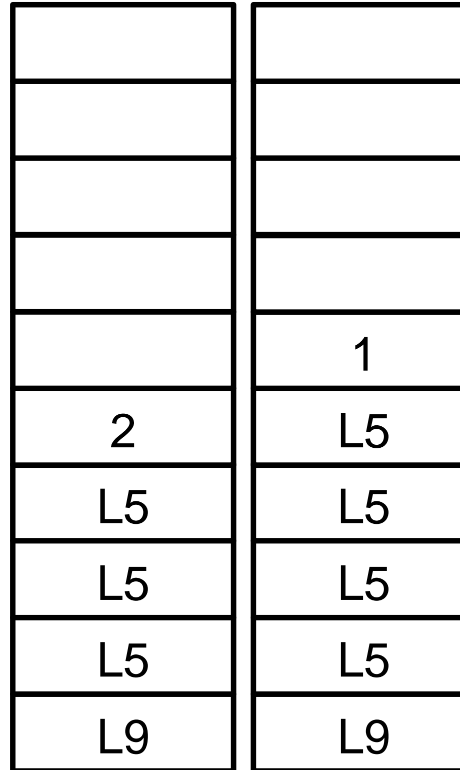

```
1  function ret = factorial(n
2          if n == 0
3                  ret = 1
4          else
5      ret = n * factorial(n−1)
6          end
7  end
8
9  factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Now facing line 1,

- Similar as before



Before After

```
1 function ret = factorial(n
2         if n == 0
3                 ret = 1
4         else
5    ret = n * factorial(n−1)
6         end
7 end
8
9 factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Now facing line 1,

- Similar as before

Before	After
	0
1	L5
L5	L5
L5	L5
L5	L5
L5	L5
L9	L9

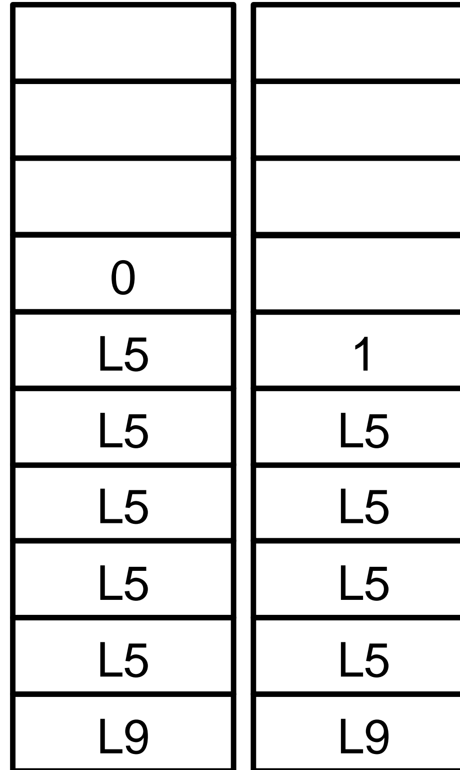
```
1  function ret = factorial(n)
2          if n == 0
3                  ret = 1
4          else
5      ret = n * factorial(n-1)
6          end
7  end
8
9  factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Now facing line 1,

- Now we get n=0 from the stack
- Execute through line 3
- Pick jump back address (top L5)
- Push return value
- Jump!



Before After

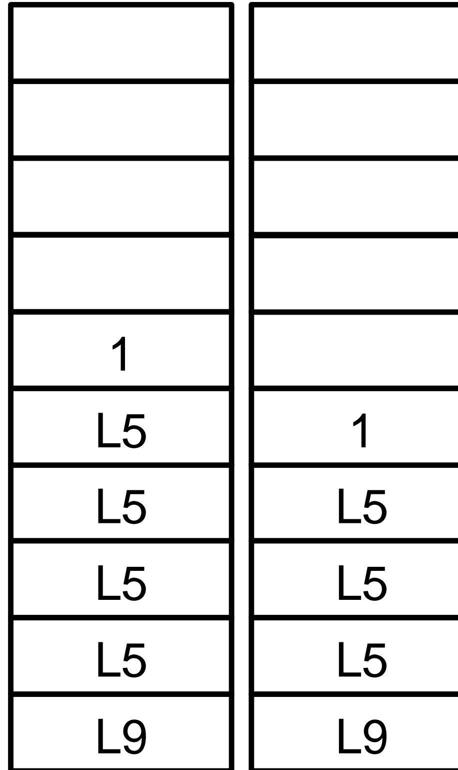
```
1 function ret = factorial(n
2         if n == 0
3                 ret = 1
4         else
5     ret = n * factorial(n-1)
6         end
7 end
8
9 factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Now facing line 5 (jumping back),

- Get ret = 1 from stack
- Execute Line 5 substituting call to factorial(0) with 1
- Pick jump back address (top L5)
- Push return value being 1*factorial(1)=1
- Jump!



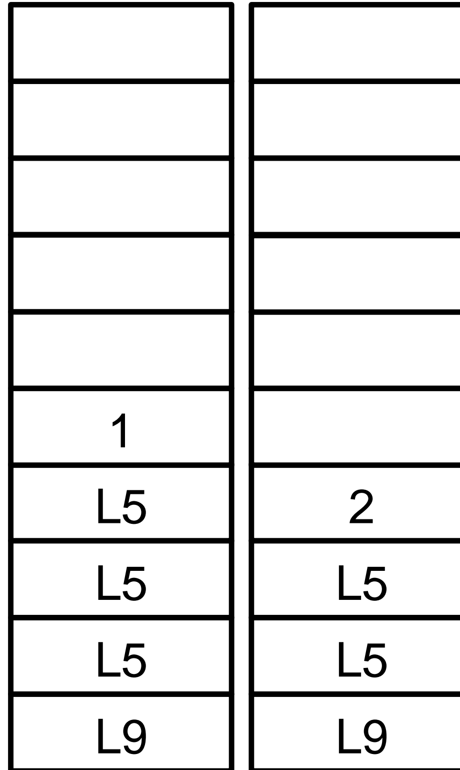

```
1  function ret = factorial(n)
2          if n == 0
3                  ret = 1
4          else
5      ret = n * factorial(n-1)
6          end
7  end
8
9  factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Now facing line 5 (jumping back),

- Get ret = 2 from stack
- Execute Line 5 substituting call to factorial(1) with 1
- Pick jump back address (top L5)
- Push return value being 2*1=2
- Jump!



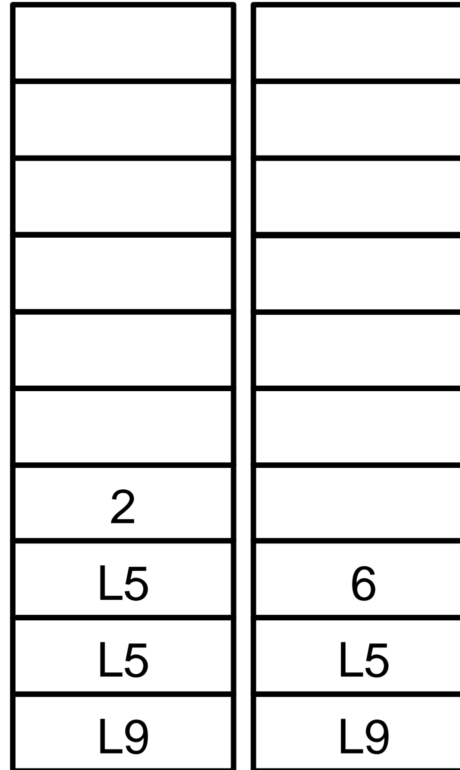

```
1  function ret = factorial(n)
2          if n == 0
3                  ret = 1
4          else
5      ret = n * factorial(n-1)
6          end
7  end
8
9  factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Similarly:

- Pop computed function value
- Compute expression
- Pop location for jump
- Push expression value as return value
- Jump!



Before After

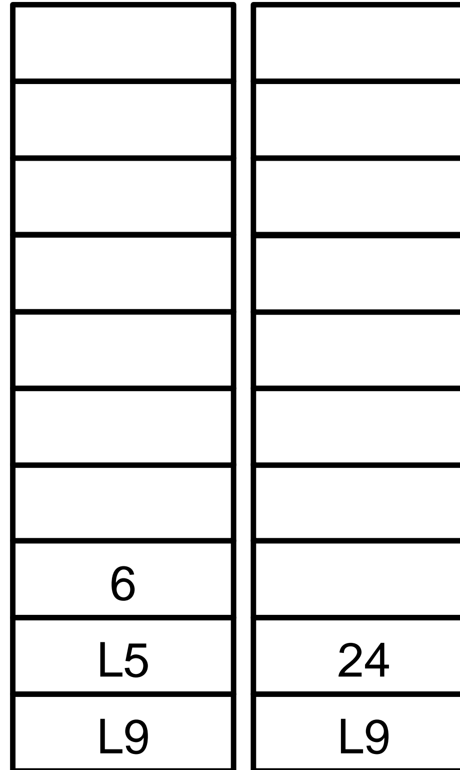
```
1 function ret = factorial(n
2         if n == 0
3                 ret = 1
4         else
5    ret = n * factorial(n−1)
6         end
7 end
8
9 factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Similarly:

- Pop computed function value
- Compute expression
- Pop location for jump
- Push expression value as return value
- Jump!



Before After

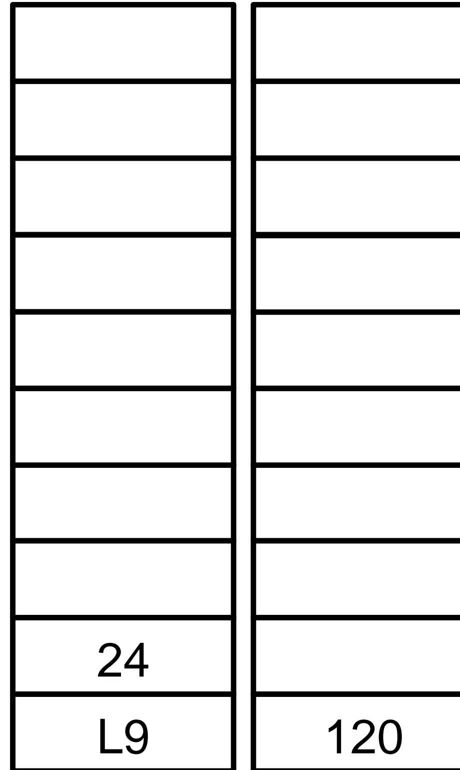
```
1 function ret = factorial(n
2         if n == 0
3                 ret = 1
4         else
5     ret = n * factorial(n-1)
6         end
7 end
8
9 factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Similarly:

- Pop computed function value
- Compute expression
- Pop location for jump
- Push expression value as return value
- Jump!

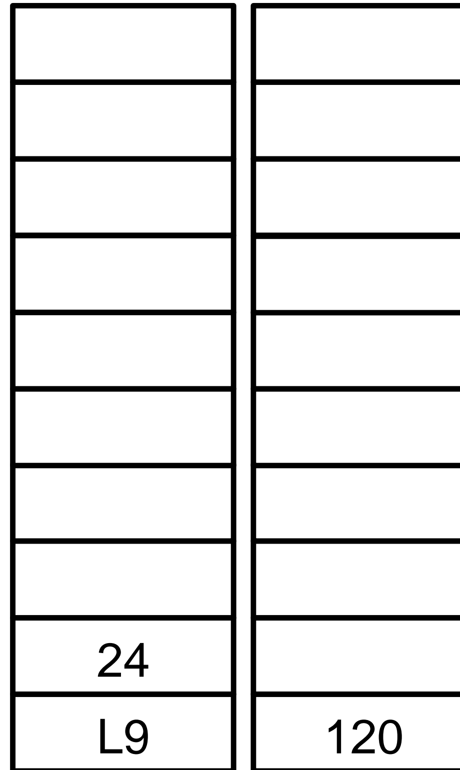



```
1  function ret = factorial(n
2          if n == 0
3                  ret = 1
4          else
5      ret = n * factorial(n-1)
6          end
7  end
8
9  factorial(5)
```

Execution of Recursion with a Stack

Trace of Execution:

Now the call stack of the recursion has been completely used. As there are no outstanding calls, the remaining value (last pushed) is the return value of the function. And as we don‘t jump to line 5 anymore, but to line 9, the program will continue with the execution of line 9 by printing the return value 120 as found on the stack.




```
1  function ret = factorial(n
2          if n == 0
3                  ret = 1
4          else
5      ret = n * factorial(n−1)
6          end
7  end
8
9  factorial(5)
```

Remark

In real computers, code locations are integer numbers just like integer numbers and other things are represented as sequences of bytes (small integer numbers) anyways. That is, there is no distinction between jump address data or attribute data. Everything is just in terms of bytes.

The signature (parameter and return value declaration) define how the stack is used by the function:

- How much to take before executing
- How much to put for the return value

Therefore, the function can safely identify the location of the jump address and jump.

In addition, note that variable-length data is a problem here which is why we will introduce the concept of pointers while looking at C++. In short, if attributes have a variable length, they are not given as parameters. Instead, their memory location is given as a pointer (just as the return address is a memory pointer, technically speaking).

This is introduced later, maybe the sentence helps when recaping at the end of the semester!