

15.3 Using coroutines to describe algorithms

Description

In this section, we show examples of how to use coroutines to describe a certain class of algorithms often referred to as *interlocked sequential execution stacks*.

A *generator* is a coroutine capable of producing a sequence of values. A new value is produced for each resume of the coroutine. The next value depends on the sequence of previously generated values. We show how to define generators for computing the mathematical functions factorial. The main purpose of the factorial example is to illustrate how coroutines work.

We assume the reader is familiar with the factorial function, but here is a short description: The factorial of a non-negative integer N , denoted by $N!$, is the product of all positive integers less than or equal to N . First we show how to define factorial as a simple method:

```
FactorialFunction(N: var integer) -> F: var integer:
  F := N
  loop: do
    if (N > 1) :then
      N := N - 1
      F := F * N
      restart(loop)
```

As can be seen, `FactorialFunction` computes the product $N * (N - 1) * (N - 2) * \dots * 2 * 1$.

Next we show how to define a generator that produce the sequence of factorial, $1!, 2!, 3!, \dots, N!, \dots$. When the coroutine is resumed, it returns the next value in the sequence when it suspends.

```
PlainFactorial: obj
  getNext -> R: var integer:
    resume(PlainFactorial)
    R := F
  F: var integer
  N: var integer
  F := 1
  cycle
    PlainFactorial.suspend
    N := N + 1
    F := F * N
```

When `PlainFactorial` is generated it executes statements until the first execution of `PlainFactorial.suspend`. A subsequent resume of `PlainFactorial` will continue execution after this suspend.

`PlainFactorial.getNext` will return the factorial of 1 which is 1. Subsequent resumes will return the next factorial as shown below:

```
UsingPlainFactorial: obj
  PlainFactorial: obj
  --
  V: var integer
  V := -- V = 1
  V := -- V = 2
  V := -- V = 6
  V := -- V = 24
```

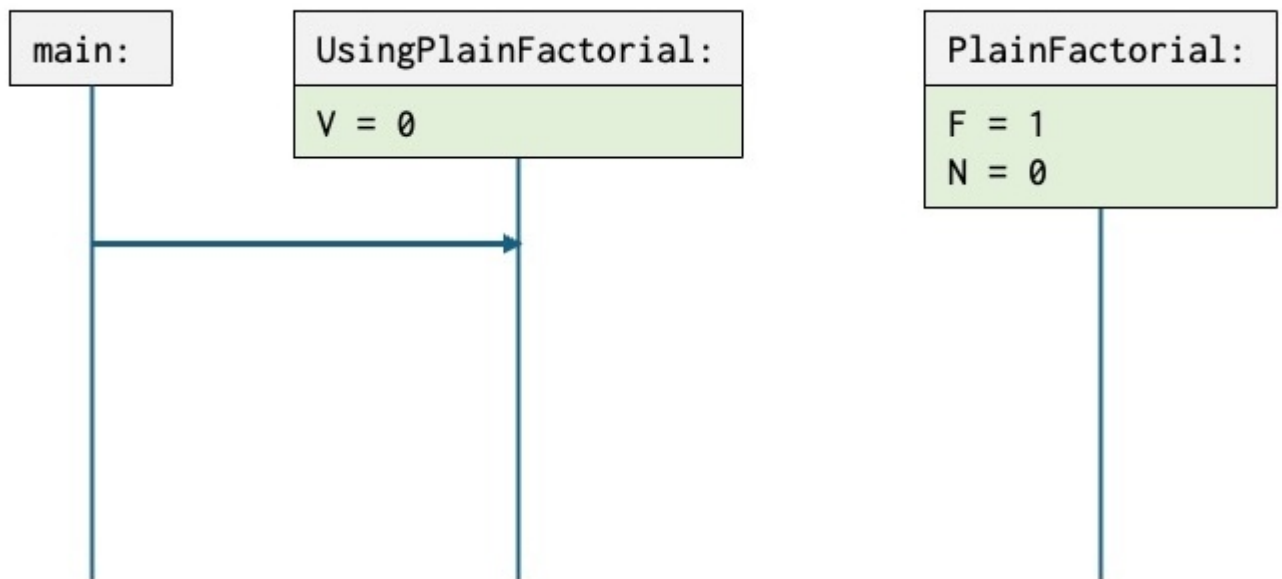
The next figure shows a snapshot of the execution of `UsingPlainFactorial` where:

- `PlainFactorial` has been generated and suspended execution during the first iteration of `cycle`.

- The read arrow \rightarrow shows that `PlainFactorial` is suspended just before the statement `N := N + 1`.
- The point of execution in `UsingPlainFactorial` is before the first statement `V := PlainFactorial.next` as indicated by the read arrow \Rightarrow .
- A read arrow of the form \Rightarrow shows the active point of execution.
- A read arrow of the form \rightarrow show the point of suspension of a coroutine.

The OSD in the right column illustrates what `UsingPlainFactorial` is currently executing and that `PlainFactorial` is suspended.

```
UsingPlainFactorial: obj
  PlainFactorial: obj
  "-"
  F := 1
  cycle
    PlainFactorial.suspend
-->   N := N + 1
      F := F * N
  V: var integer
=> V :=
  V :=
  V :=
  V :=
```



The OSD also shows the values of the variable `V` in `UsingPlainFactorial`, and the variables `F` and `N` in `PlainFactorial`.

The next next scenario shows the situation:

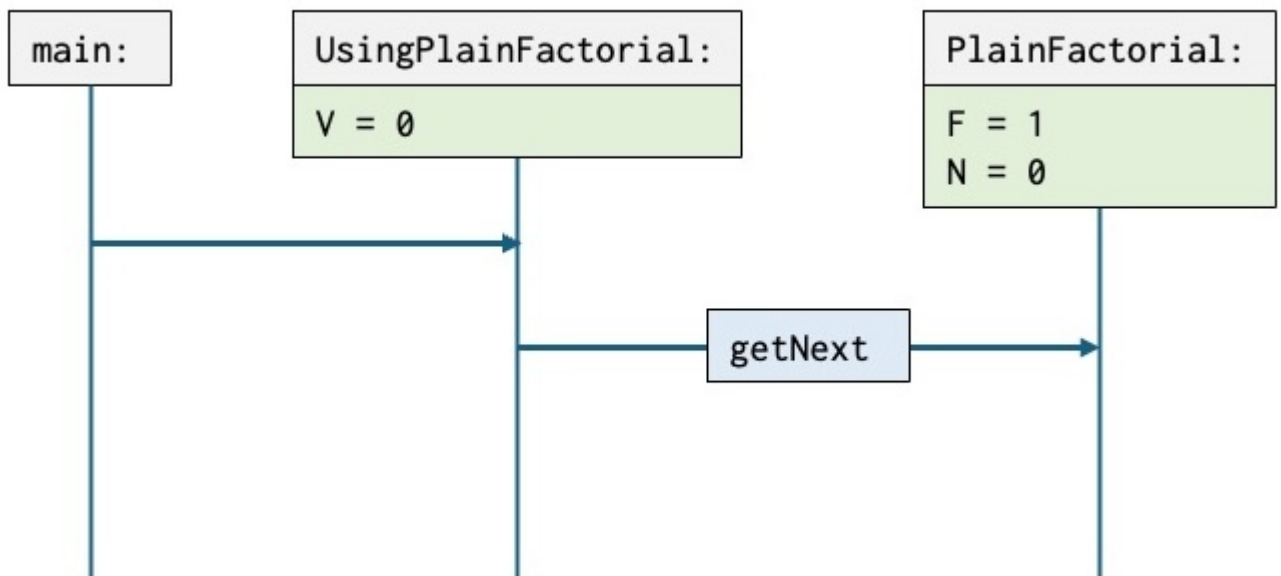
- `PlainFactorial.getNext` has been executed by `UsingPlainFactorial`.
- `getNext` has executed `resume(PlainFactorial)`.
- The next statement to be executed is `N := N + 1`.

```
UsingPlainFactorial: obj
  PlainFactorial: obj
  getNext -> R: var integer:
    resume(PlainFactorial)
  R := F
```

```

F: var integer
N: var integer
F := 1
cycle
  PlainFactorial.suspend
==>   N := N + 1
      F := F * N
V: var integer
V :=
V :=
V :=
V :=

```



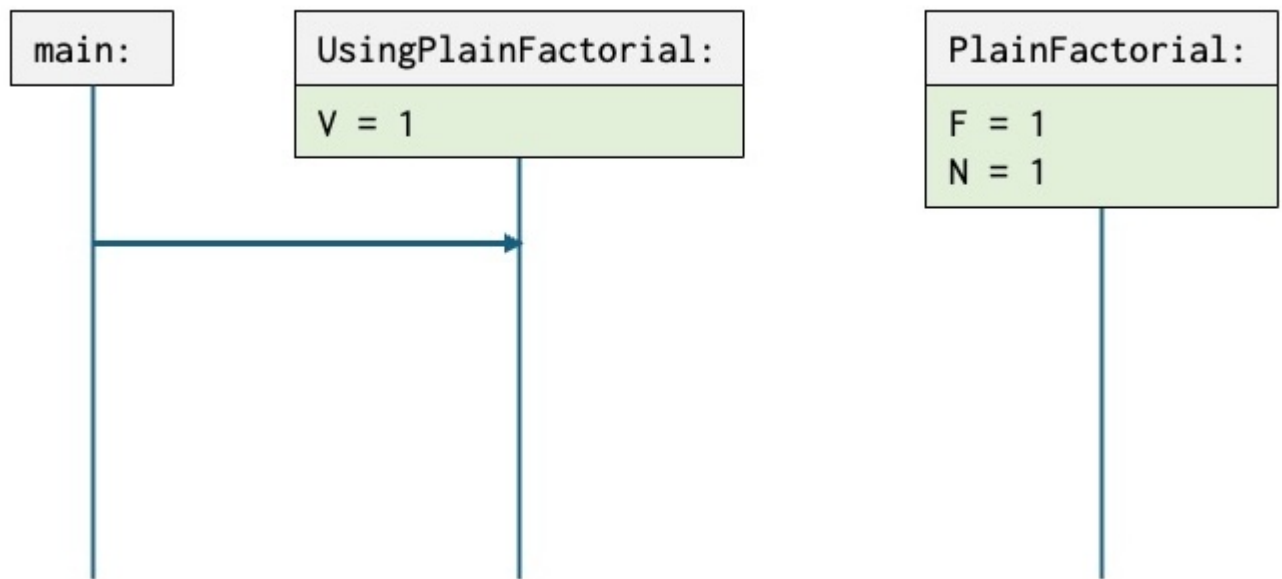
The next snapshot shows the situation after `PlainFactorial` has executed the two assignments statements, `N := N + 1` and `F := F * N`, and executed a `PlainFactorial.suspend` during the second iteration of `cycle`:

- `PlainFactorial` is suspended before the statement `N := N + 1` – as in the first figure above.
- The active point of execution is in `UsingPlainFactorial` before the second statement `V := PlainFactorial.getNext`.
- The variables have been updated `V`, `F` and `N` have all new values.

```

UsingPlainFactorial: obj
  PlainFactorial: obj
  "-"
  cycle
    PlainFactorial.suspend
-->   N := N + 1
      F := F * N
V: var integer
V :=
==> V :=
    V :=
    V :=

```



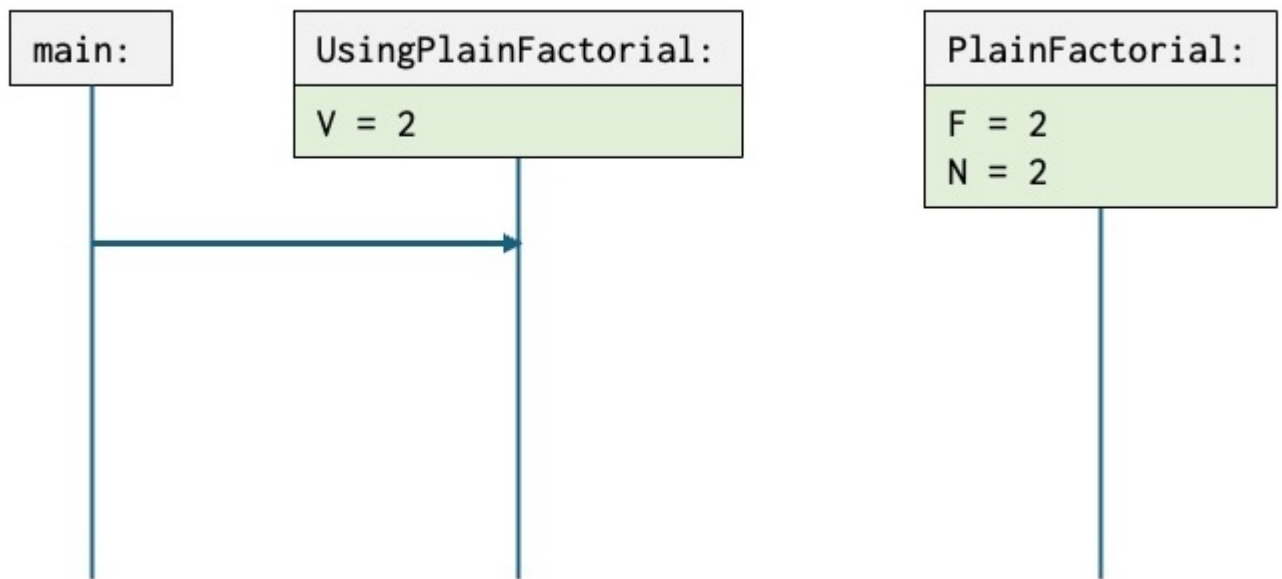
Execution of the second `V :=` resumes execution of `PlainFactorial`, which then generate the next factorial value. We don't show a snapshot of this.

The next snapshot shows the situation after `PlainFactorial` has suspended:

- `PlainFactorial` is as in previous snapshots suspended before the statement `N := N + 1`.
- The current point of execution is in `UsingPlainFactorial` before the third statement `V := PlainFactorial.getNext`.
- The variables `V`, `F` and `N` have been updated.

```

UsingPlainFactorial: obj
PlainFactorial: obj
  "-"
  cycle
    PlainFactorial.suspend
-->    N := N + 1
      F := F * N
V: var integer
V :=
V :=
==> V :=
     V :=
    
```

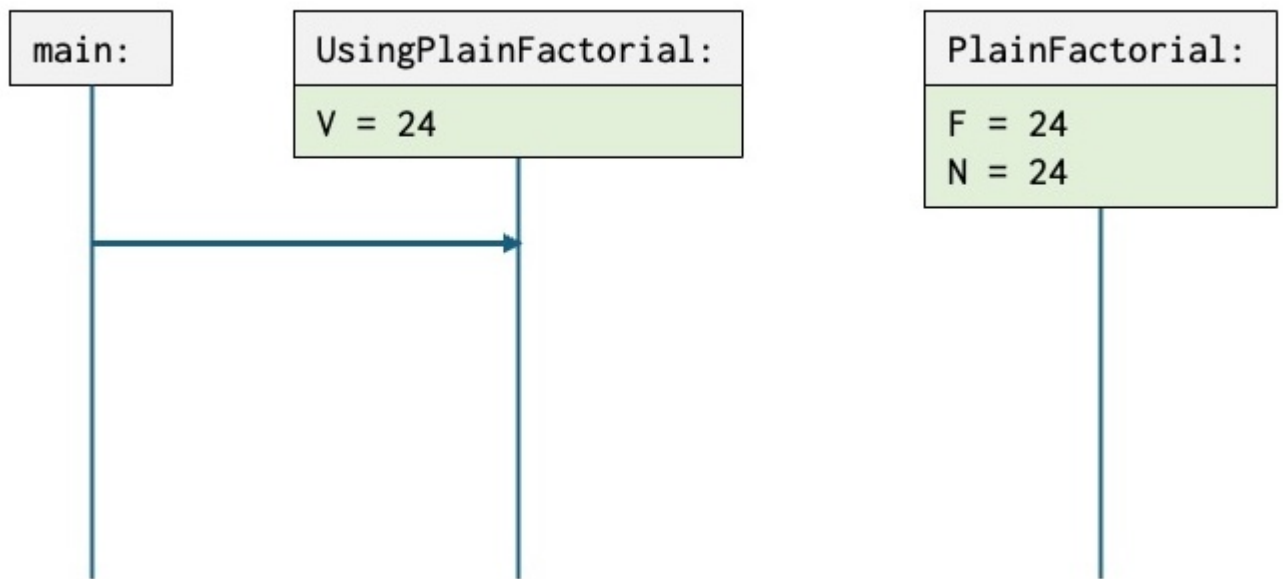


Finally we show a snapshot at the situation after execution of the last `V := PlainFactorial.getNext`:

- PlainFactorial is (again) suspended before the statement `N := N + 1`.
- The active point of execution is after the last statement.
- The variables `V`, `F` and `N` have been updated.
- `V = 24 = 6!`

```

UsingPlainFactorial: obj
  PlainFactorial: obj
    -"-
    cycle
      PlainFactorial.suspend
-->      N := N + 1
          F := F * N
  V: var integer
  V :=
  V :=
  V :=
  V :=
==>
    
```



Recursive factorial generator

Next we show a version of a factorial generator using a recursive method instead of the loop implemented using `cycle`. As said the purpose is to show how coroutines work and not necessarily a recommended programming style.

```
RecursiveFactorial: obj
  getNext -> R: var integer:
    resume(RecursiveFactorial)
    R := F
  F: var integer
  N: var integer
  next:
    RecursiveFactorial.suspend
    N := N + 1
    F := F * N
    next
  F := 1
  next
```

When `RecursiveFactorial` is generated it invokes the local method `next`, which suspends execution. Successive invocations of `next` resumes the coroutine and returns the next factorial (F) in the sequence.

The next figures shows snapshots of using `RecursiveFactorial` by the object `UsingRecursiveFactorial`:

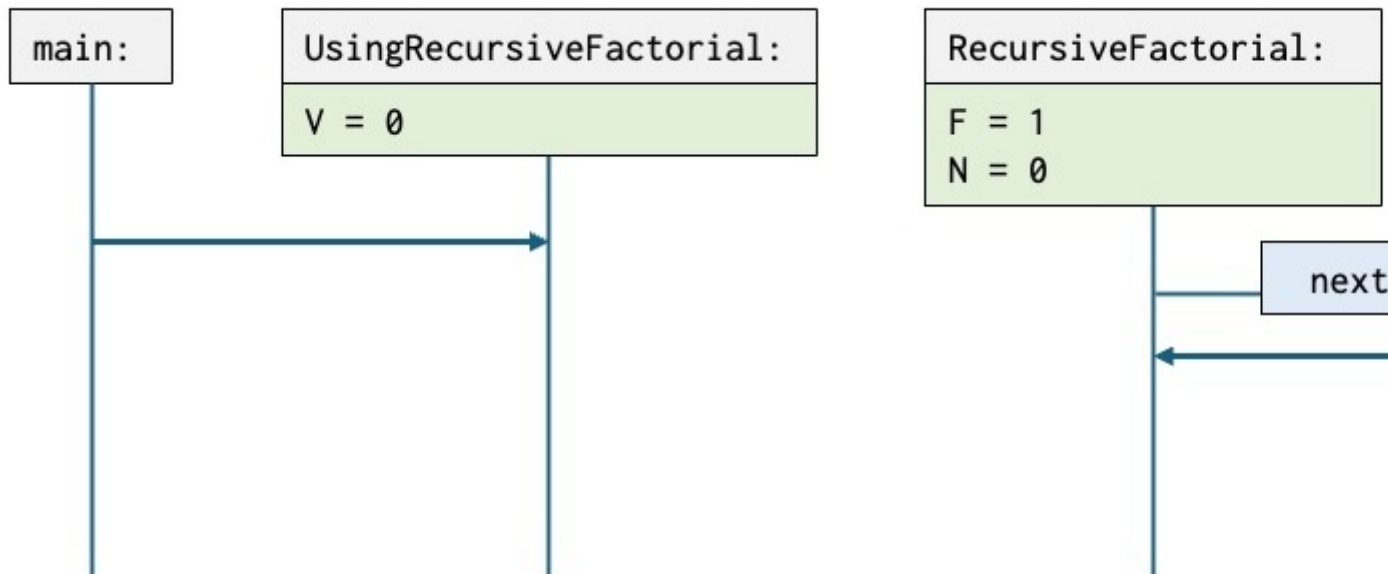
```
UsingRecursiveFactorial: obj
  RecursiveFactorial: obj
  "--
  V: var integer
  V := Recursive -- V = 1
  V := Recursive -- V = 2
  V := Recursive -- V = 6
  V := Recursive -- V = 24
```

The first snapshot how the situation after the objects `UsingRecursiveFactorial` and `RecursiveFactorial` have been generated:

- RecursiveFactorial is suspended before $N := N + 1$ in the method object next.
- Note that RecursiveFactorial has been suspended while execution an instance of the method next.
- The active point of execution is at the first $V := RecursiveFactorial.getNext$ in UsingRecursiveFactorial.
- The values of the variables are $V = 0, F = 1$ and $N = 0$.

```

UsingRecursiveFactorial: obj
RecursiveFactorial: obj
  getNext -> R: var integer:
    resume(RecursiveFactorial)
    R := F
  F: var integer
  N: var integer
  next:
    RecursiveFactorial.suspend
-->    N := N + 1
        F := F * N
        next
    F := 1
    next
V: var integer
==> V := Recursive
    V := Recursive
    V := Recursive
    V := Recursive
    
```



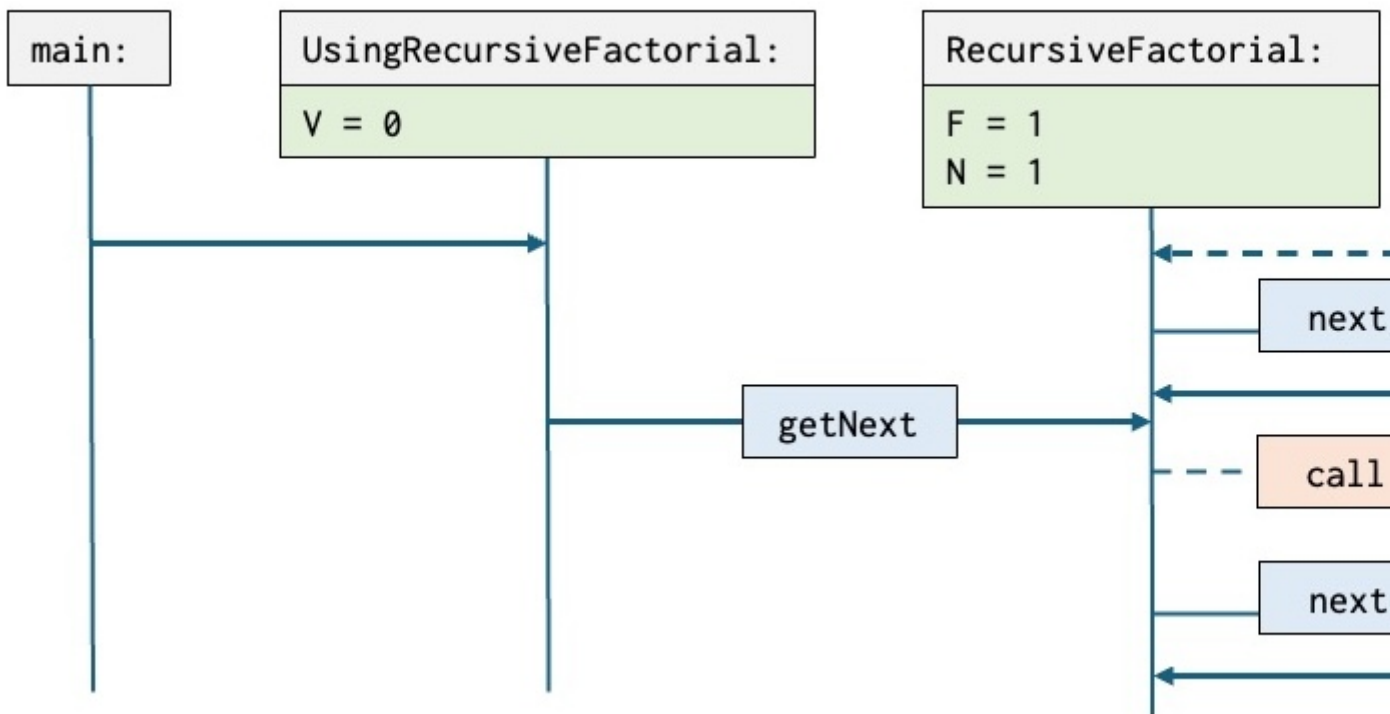
The next snapshot shows a situation after execution of the first `RecursiveFactorial.getNext`:

- RecursiveFactorial has been resumed.
- $N := N + 1$ has been executed — $N = 1$.
- $F := F * N$ has been executed — $F = 1$.
- A recursive invocation of next has been generated.

- The active point of execution is at RecursiveFactorial.suspend in this next method.

```

UsingRecursiveFactorial: obj
  RecursiveFactorial: obj
  "-"
  next:
==>    RecursiveFactorial.suspend
        N := N + 1
        F := F * N
        next
  F := 1
  next
V: var integer
V := Recursive
V := Recursive
V := Recursive
V := Recursive
V := Recursive
    
```



```

RecFactorial: obj
  out V: var integer
  N: var integer
  next:
    RecFactorial.suspend
    N := N + 1
    V := V * N
    next
  V := 1
  next
    
```

```

X: var integer
X := RecFactorial -- X = 1
X := RecFactorial -- X = 2
X := RecFactorial -- X = 6
X := RecFactorial -- X = 24
    
```

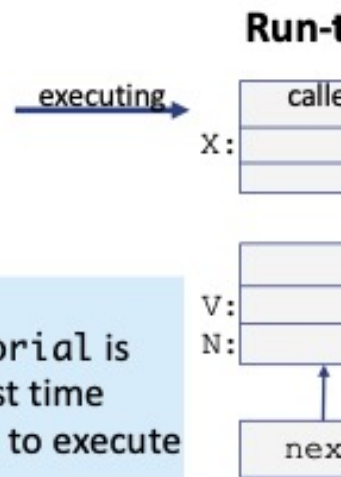
```

V := Recursive -- V = 2

V := Recursive -- V = 6

V := Recursive -- V = 24
    
```

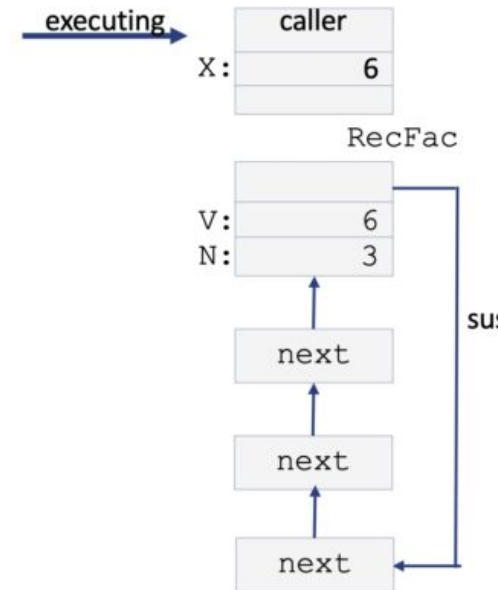
- Initial state
- RecursiveFactorial is suspended the first time
- The caller is ready to execute first call of RecursiveFactorial



```

RecFactorial: obj
  out V: var integer
  N: var integer
  next:
    RecFactorial.suspend
    N := N + 1
    V := V * N
    next
  V := 1
  next
X: var integer
X := RecFactorial -- X = 1
X := RecFactorial -- X = 2
X := RecFactorial -- X = 6
X := RecFactorial -- X = 24
    
```

Run-time snapshots:



As can be seen, a suspend within next, suspends the whole execution stack and a resume resumes execution of the top element of the stack.

Merging binary search trees

The next example is more interesting. Here we show how to merge two binary search trees. We assume that the reader is familiar with the concept of a binary search tree +++ evt henvisinging.

In this example, we define a binary tree where the a node contains a String being the name of a person. First we define class BinarySearchTree:

```

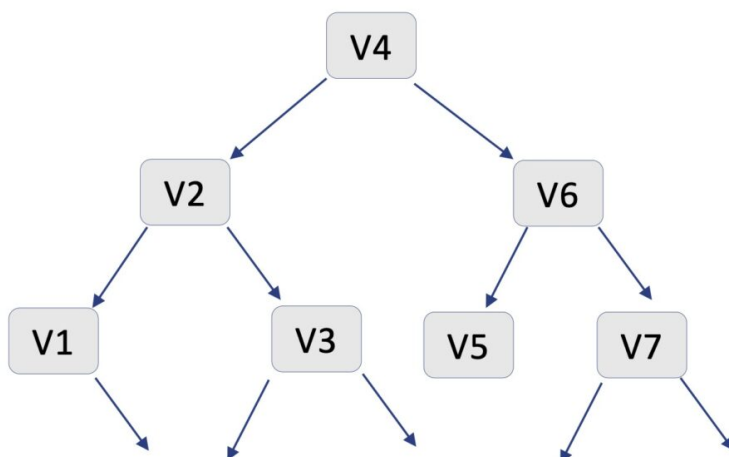
class BinaryTree:
  class node(elm: var String):
    left: ref node
    right: ref node
    insert(S: var String):
      if (S <= elm) :then
        if (left == none) :then
          left := node(S)
        :else
          left.insert(S)
      :else
        if (right == none) :then
          right := node(S)
        :else
          right.insert(S)
    print(ind: var integer):
      ...
  root: ref node
  insert(S: var string):
    if (root == none) :then
      root := node(S)
    :else
      root.insert(S)
  :::
    
```

A `BinaryTree` has the following attributes:

1. A local class `Node`.
2. A reference `root`, that refers to the root (top `Node`) of the tree.
3. A method `insert` for inserting a `Node` in the tree with the parameter `S` being the `String` stored in the `Node`. It works as follows:
 - If `root == none`, then the insertion is the first `Node` else `root.insert(S)` is invoked.
4. A `Node` has the following attributes:
5. References `left` and `right` that refer to the left and right branches of the `Node` respectively.
6. An `insert` method that works as follows:
 - If `S <= elm` where `elm` is the name in the current `Node`, then `S` is inserted in the left branch.
 - If `left == none` then `S` is the first element in the left branch and `left := Node(S)` is executed otherwise `left.insert(S)` is executed recursively.
 - If `S > elm` then `S` is inserted in the right branch.
7. A print method not shown.

When a `Node` is inserted into the tree it is ordered based on a lexicographical ordering. I.e. "Dave" comes before "John" ("Dave" < "John").

Preliminary figure



We may then declare two `BinaryTree` objects and insert some elements into them:

```

boys: obj BinaryTree
girls: obj BinaryTree
boys.insert("Peter")
girls.insert("Cecilie")
girls.insert("Maria")
boys.insert("Robin")
...

```

Next we add a coroutine, `theScanner`, that traverse the tree and for each node in the tree, it suspends execution and returns the value at the node. If the tree has `n` nodes it returns a sequence of `String` values where `V1 <= V2 <= V3 <= ... <= Vn`.

```

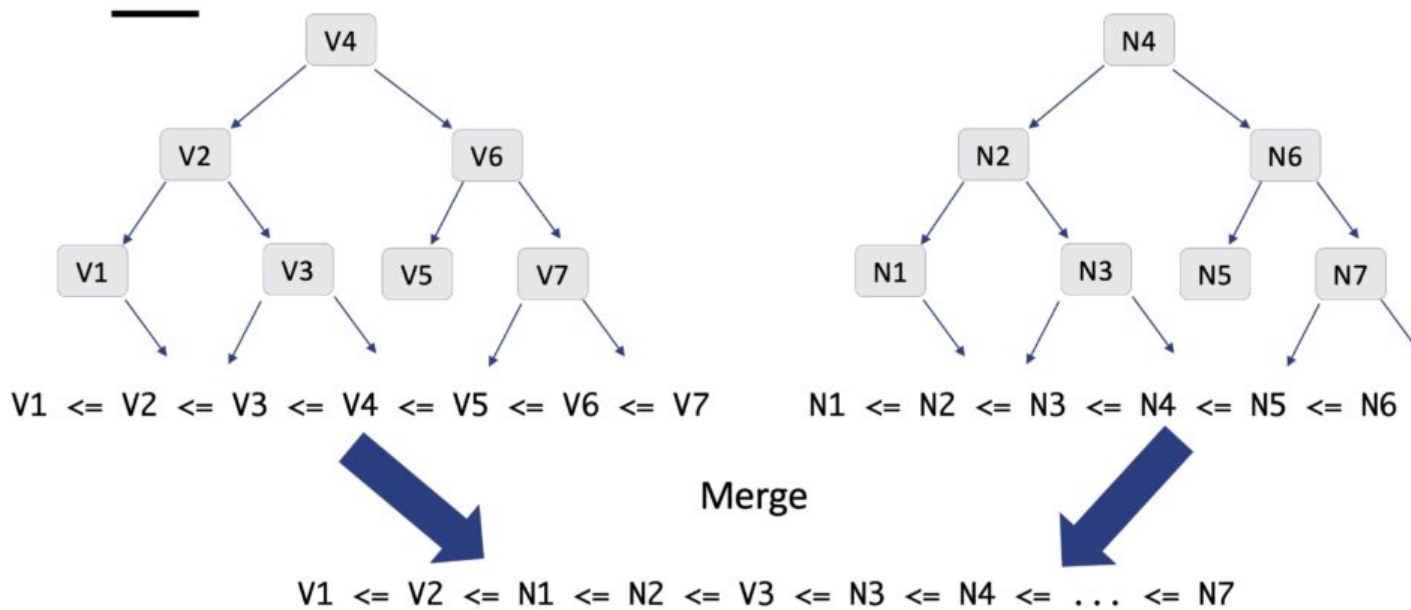
class BinaryTree:
    -"-
    theScanner: obj
        CV: var String
        next -> V: var String:

```

```
        resume(theScanner)
        V := CV
    scan(current: ref node):
        if (current /= none) :then
            scan(current.left)
            CV := current.elm
            theScanner.suspend
            scan(current.right)
        theScanner.suspend
    scan(root)
    CV := ""
    theScanner.suspend
next -> V: var String:
    V := theScanner.next
```

Finally we may add a method, merge that merges the values of two binary search trees:

```
merge:
    nextBoy: var String
    nextGirl: var String
    tail(T: ref BinaryTree):
        cycle
            S: var String
            S := T.next
            if (S = "") :then
                leave(tail)
            :else
                S.print
    nextBoy := boys.next
    nextGirl := girls.next
    loop: do
        if (nextBoy <= nextGirl) :then
            if (nextBoy = "") :then
                nextGirl.print
                tail(girls)
            :else
                nextBoy.print
                nextBoy := boys.next
                restart(loop)
        :else
            -- nextBoy > nextGirl
            if (nextGirl = "") :then
                nextBoy.print
                tail(boys)
            :else
                nextGirl.print
                if (nextBoy.length > 0) :then
                    nextGirl := girls.next
                restart(loop)
```



The merge method works as follows:

1. The statements: `nextBoy := boys.next` and `nextGirl := girls.next` assigns the first boy and first girl to `nextBoy` and `nextGirl` respectively where the ordering is alphabetical.
2. If `nextBoy <= nextGirl`, then `nextBoy` is printed and `nextBoy` is assigned the next boy from the tree using `boys.next`.
3. If `nextBoy > nextGirl`, then `nextGirl` is printed and assigned the next girl.
4. This is repeated until no more boys and girls in the trees.
5. The termination condition is that the empty string (" ") is returned by `next` if no more girls/boys in a tree.
6. if the boys tree becomes empty before the girls tree, then the method invocation `tail(girls)` prints the remaining girls in the tree – similarly if the girls tree becomes empty before the boys tree.