Part V.
Programming Examples

Examples of OOCP Concepts and Methodology through Programming Examples in Actalk

Concept(s)	*Example*
Continuation	Factorial
Divide & Conquer and Join Continuation	MultiplyInRange
Pipeline	Prime Numbers
Synchronization	Bounded Buffer

Computing Factorial with Continuations

Recursive computation of **factorial(n)** is split between:

* the recursive call **factorial(n-1)**,

* multiplying that value by **n** and returning it (to the current reply destination)

This latter task is delegated to a continuation which encapsulates:

\# the program of the remaining computation

(multiply **factorial(n-1)** by **n**, and return the result to current reply destination),

\# the context of the current computation needed to resume it later

(**n** and the current reply destination)

Factorial in Scheme (lexical scope Lisp)

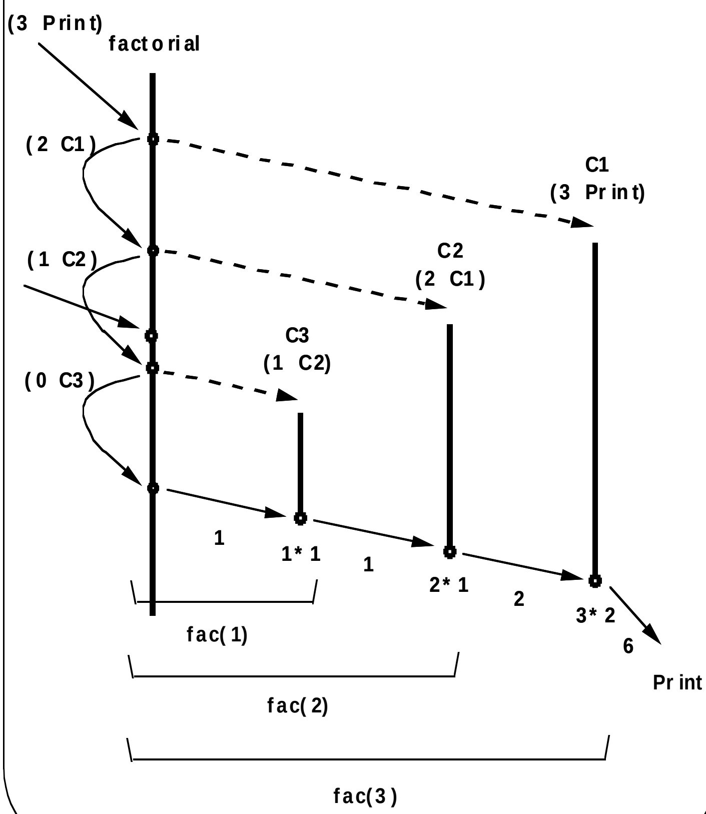
```
; factorial with recursion

(define (factorial-r n)
  (if (= n 0)
    1
    (* (factorial-r (- n 1)) n)))
```

```
? (factorial-r 10)
= 3628800
```

```
; factorial with continuation

(define (factorial-c n c)
  (if (= n 0)
    (c 1)
    (factorial-c (- n 1)
                 (lambda (v) (c (* v n))))))
```

```
? (factorial-c 10 (lambda (v) v))
= 3628800
```


Factorial



class Factorial

```
ActiveObject: #Factorial
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!Factorial methodsFor: 'script'!

n: n replyTo: r
  n = 0
    ifTrue: [r reply: 1]
    ifFalse: [aself n: n-1 replyTo:
      (FactorialContinuation new n: n r: r) active]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --"!

!Factorial class methodsFor: 'example'!

example
  "Factorial example"
  | aFac |
  aFac := Factorial new active.
  aFac n: 10 replyTo: Print.
  aFac n: 5 replyTo: Print! !
```

Print is a predefined active object which represents the display (Transcript window)

class FactorialContinuation

```
ActiveObject subclass: #FactorialContinuation
  instanceVariableNames: 'n r '
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!FactorialContinuation methodsFor: 'initialization'!

n: anInteger r: aContinuation
  n := anInteger.
  r := aContinuation! !

!FactorialContinuation methodsFor: 'script'!

reply: v
  r reply: n * v! !
```


Advantages of Continuation

<u>asynchronous + continuation</u>

vs synchronous call

in case of divergence

ex: factorial(-1)

availability remains,

but eventually no more resources (memory full!)

recursion is possible

with synchronous communication, recursion implies deadlock!

However systematic programming with continuations is too low level. They may be automatically generated from a higher level language by a compiler (actor approach, see Part VII).

Levels of Concurrency

In fact the computation of factorial actually remains sequential!

Concurrency occurs within simultaneous requests

We will now describe another algorithm managing concurrent sub-computations

The idea is to decompose computation into sub-computations executed concurrently with recombination of partial sub-results

Factorial with Recursive Partition

Computation of N! is equivalent to multiplying all numbers in interval [1 N]

These multiplications may occur concurrently. The idea is to divide the interval [1 N] into two sub-intervals:

[1 mid] and [mid+1 N]

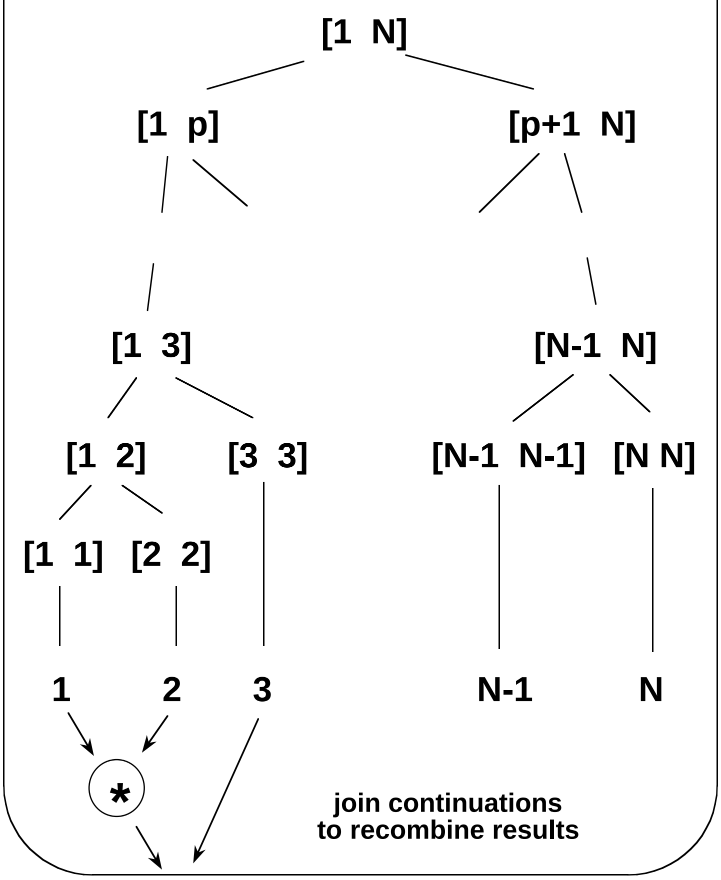
(where mid = 1+N//2)

and to perform the two sub-computations concurrently

Computation is recursive and creates two new active objects to compute these two sub-intervals with a <u>join continuation</u> as the common reply destination

This join continuation will multiply the two partial results

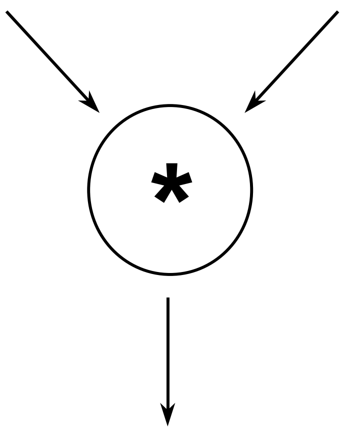
Factorial with Divide and Conquer



Divide and Conquer and Join Continuations

Join continuations resynchronize sub-computations and recombine partial results

A join continuation is equivalent to a data-flow operator, which waits for two (or more) values to recombine, and sends the result to the current continuation



continuation

class MultiplyInRange

```
ActiveObject subclass: #MultiplyInRange
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: ' Tutorial-Examples'!


!MultiplyInRange methodsFor: 'script'!

from: i to: n replyTo: r
   | jc mid |                          "[i  n]"
   i = n ifTrue: [r reply: i]
      ifFalse:
      [jc := (MultiplyJoinContinuation new r: r) active.
       mid := (i+n)//2.
       MultiplyInRange new        "[i  mid]"
          from: i to: mid replyTo: jc.
       MultiplyInRange new        "[mid+1  n]"
          from: mid+1 to: n replyTo: jc]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

MultiplyInRange class
   instanceVariableNames: ''!


!MultiplyInRange class methodsFor: 'example'!

example
   "MultiplyInRange example"
   MultiplyInRange new active
      from: 1 to: 10 replyTo: Print! !
```

class MultiplyJoinContinuation

```
ActiveObject subclass: #MultiplyJoinContinuation
   instanceVariableNames: 'v1 c '
   classVariableNames: ''
   poolDictionaries: ''
   category: ' Tutorial-Examples'!


!MultiplyJoinContinuation methodsFor: 'initialization'!

c: aContinuation
   c := aContinuation! !

!MultiplyJoinContinuation methodsFor: 'script'!

reply: v
   v1 isNil              "if no value received yet"
      ifTrue: [v1 := v] "memorize first value"
      ifFalse: [c reply: v1*v]! ! "otherwise, compute"
```

The behavior of the join continuation changes serially:

1) memorize value

2) combine (multiply) it with the first value and return the result to the continuation

A test is necessary to check if a first value has already been accepted. This problem will be solved gracefully in the actor model of computation (see Part VII).

Concurrent Computation/ Problem Solving

Three main classes of concurrent algorithms for problem solving are:

\# <u>partition</u>

the problem is decomposed in sub-problems, partial results are recombined afterwards

\# <u>pipeline</u>

the problem is decomposed by linking up computations through which data flow up

\# <u>cooperation</u>

the problem is decomposed in a collection of entities (agents) which will themselves organize decomposition, allocation, and coordination of their tasks *(see multi-agent systems, Part IX)*

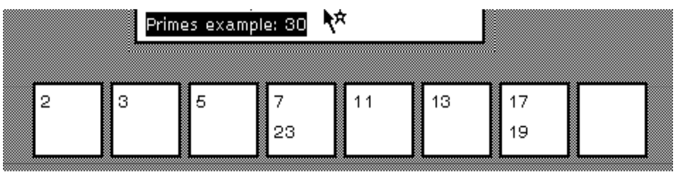
Pipeline: Computation of Prime Numbers

We compute prime numbers through an ordered chain of sieves

Each sieve represents a prime number already found

Successive integers are sent through the chain, each filter testing if it divides the integer

If an integer reaches successfully the last filter of the chain, a new prime number has been found, it will then be added at the end of the chain



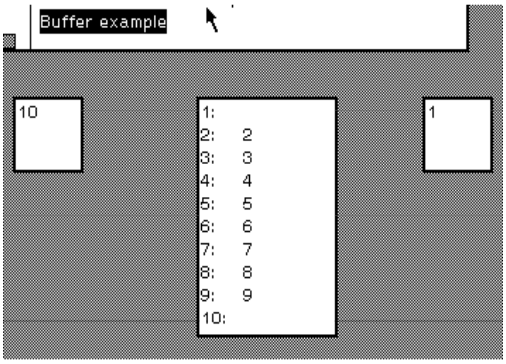
class PrimeFilter

```
ActiveObject subclass: #PrimeFilter
   instanceVariableNames: 'n next '
   classVariableNames: ''
   poolDictionaries: ''
   category: ' Tutorial-Examples'!

!PrimeFilter methodsFor: 'initializing'!

n: aPrimeNumber
   n := aPrimeNumber! !

!PrimeFilter methodsFor: 'script'!

filter: i
   i \\ n = 0                    "if i is not divided by n"
      ifFalse: [next isNil       "if end of the chain"
            "a new prime number is added to the chain"
         ifTrue: [next := (PrimeFilter new n: i) active]
            "otherwise pass the test to next in the chain"
         ifFalse: [next filter: i]]! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --"!

!PrimeFilter class methodsFor: 'example'!

checkUntil: max
   "PrimeFilter checkUntil: 50"
   | two |
   two := (PrimeFilter new n: 2) active.
   2 to: max do: [:i | two filter: i]! !
```

Synchronization: Producer/Consumer with a Bounded Buffer

A producer and consumer exchange data through a bounded buffer

The issue is to synchronize production and consumtion to the availability (fullness or emptyness) of the buffer (e.g., disable put requests while the buffer is full)



Programming Methodology and Examples in Actalk

class BoundedBufferActivity

(Abstract States model of synchronization)

```
AbstractStatesActivity subclass: #ASBBActivity
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!ASBBActivity methodsFor: 'abstract states'!

empty
  ^#(put:)!

full
  ^#(get)!

initialAbstractState
  ^#empty!

partial
  ^(self empty) + (self full)! !

!ASBBActivity methodsFor: 'state transition'!

nextAbstractStateAfter: selector
  ^oself is empty
       ifTrue: [#empty]
       ifFalse: [oself isFull
                  ifTrue: [#full]
                  ifFalse: [#partial]]! !
```


Programming Methodology and Examples in Actalk

class Producer

```
ActiveObject subclass: #Producer
  instanceVariableNames: 'buffer delay '
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!Producer methodsFor: 'initializing'!

buffer: aBoundedBuffer delay: seconds
  buffer := aBoundedBuffer.
  delay: seconds! !

!Producer methodsFor: 'script'!

run: max
  1 to: max do: [:i |
    buffer put: i.
    (Delay forSeconds: delay) wait]! !
```


Programming Methodology and Examples in Actalk

class Consumer

```
ActiveObject subclass: #Consumer
  instanceVariableNames: 'buffer delay '
  classVariableNames: ''
  poolDictionaries: ''
  category: ' Tutorial-Examples'!

!Consumer methodsFor: 'initializing'!

buffer: aBoundedBuffer delay: seconds
  buffer := aBoundedBuffer.
  delay: seconds! !

!Consumer methodsFor: 'script'!

run: max
  max timesRepeat:
    [buffer get.
     (Delay forSeconds: delay) wait]! !
```

Monitors (on Passive Objects)

activation synchronization = mutual exclusion

+ service synchronization (suspension onto conditions)

Service synchronization is expressed with conditions (variables: Hoare, or expressions: Kessel) on which processes will be suspended while conditions are not fulfilled