# Versions of Factorial

*Focal concepts:*

Each of these encodings of the *factorial function* is functionally equivalent. How they achieve the functionality differs.

Almost all are legitmate Mathematica code. Since the core process in Mma is the same for each encoding, we have a demonstration that all are *statically equivalent*. Dynamically, ie how the code runs, all are different.

The *style of encoding* should match as closely as possible the form of the natural problem. Second, the style should match the coder's natural way of thinking about the problem.

Types of *dynamic differences* include:

• *Syntactic sugar*: the same dynamic behavior (ie the same language). Macros expand the sugared notation *at read-time* into standard notation. Eg:

```
(a + b)  ==>  +[a,b]

declare a=5;  (a + b)
```

• *Functional syntactic sugar*: shorter and specialized versions of functions. The compiler usually standardizes these variants. Eg, all of the various loop constructs are the same.

```
for i=1 to n do Process[i]

i:=0; (do Process[i]; i:=i+1 until i=n)

dotimes[n, Process[#]]

StreamProcess[IntegerStream[1, n]]
```

• *Functional model difference*: different processes for achieving the same functional objective. Most of these compile into different machine instructions, but a good optimizing compiler might standardize some of them. Eg: iteration vs recursion vs mapping

```
do[i from 1 to n, acc from nil, Process[i, acc]]

(if i=n, acc, Process[i-1, F[acc, i]])

(if i=n, 0, F[i, Process[i-1]])

map[Process, {1,i,n}]
```

- ***Operational difference***: different engines achieve the same objective but use different operational characteristics. Eg:

```
F[1]=1; F[n]= G[n, F[n-1]]

(if test[n] then (res:=F[i], ++i) else res)

(send F, n)
```

- ***Mathematical difference***: different mathematical computations achieve the same objective but use different models. Eg:

```
F[n] = G[n]             eg Fac[n]=Gamma[n+1]

Decode[Process[Encode[F,n]]]

When (F[Guess[n1] - F[Guess[n2]] = <small>), F[n1]
```

- ***Level of Implementation difference***: different processes occur at different levels of abstraction. Eg:

```
2 + 5 = 7

010 + 101 = 111

r1=Load[i0]; r2=Fetch[j0]; r3=Add[r1,r2]; Store[r3]

b0 = xor[i0,j0]; b[1] = xor[i1,j1]
```

## VERSIONS

```
1.  proceduralFactorial[n] :=
      if ( Integer[n] and Positive[n] )
            then
                  Block[ {iterator = n,
                         result = 1  },
                    While[ iterator != 1,
                        result := result * iterator;
                        iterator := iterator - 1 ];
                    return result]
            else  Error
```

```
2.  sugaredProceduralFactorial[n] :=
      Block[  {result = 1},
          Do[  result = result * i, {i, 1, n} ];
          result]
```

```
3.  loopFactorial[n] :=
      { For[ i=1 to n, i++, result := i*result ];
        result }


4.  guardedFactorial[n, result] :=
      Precondition:     Integer[n] and Positive[n]        /also end condition
      Invariant:        factorial[n] = n * factorial[n - 1]
      Body:             guardedFactorial[ (n - 1), (n * result) ]
      PostCondition:    result = Integer[result] and Positive[result]
                                    and (result >= n)


5.  assignmentFactorial[n] :=
      { product := 1;
        counter := 1;
        return assignmentFactorialCall[n, product, counter] }


6.  assignmentFactorialCall[n, product, counter] :=
      if[ (counter > n)
            then
                  return product
            else
                  { product := (counter * product);       /error if these are
                    counter := (counter + 1);             /in reverse order
                    return assignmentFactorialCall[n, product, counter] } ]


7.  recursiveFactorial[n] :=
      if[  n == 1, 1,  n*recursiveFactorial[n - 1] ]


8.  rulebasedFactorial[1] = 1;
    rulebasedFactorial[n] := n * rulebasedFactorial[n - 1]


9.  accumulatingFactorial[n, result] :=
      if[ (n = 0)
            then
                  return result
            else
                  return accumulatingFactorial[ (n - 1), (n * result) ]


10. upwardAccumulatingFactorial[product counter max] :=
      if[ (counter > max)
            then
                  return product
            else
                  return upwardAccumulatingFactorial[ (counter * product)
                                                      (counter + 1)
                                                      max ] ]
```

**11. mathematicalFactorial**[n] =
```
      Apply[ Times, Range[n] ]
```

**12. generatorFactorial**[n]
```
      Times[ i, Generator[i, 1, n] ]
```

**13. combinatorFactorial :=**
```
      Y f< n< COND (=0 n) 1 (* n (f (-1 n))) >>
```

**14. sugaredCombinatorFactorial** =
```
      S (CP COND =0 1) (S * (B FAC -1)))
```

**15. integralFactorial**[n] =  Gamma[ n + 1 ]  :=
```
      integral[ 0 to Infinity, (t^n * e^(1 - n)), dt ]
```

**16. streamOfFactorials** =
```
    streamAttach[ 1 streamTimes[streamOfFactorials streamOfPositiveIntegers] ]
streamOfPositiveIntegers =
    streamAttach[ 1 streamBuild[ Add1 CurrentStreamValue ] ]
```

**17. JamesCalculusFactorial**[n] =
```
      Decode[Standardize[Do[Stack[Encode[i], acc] {i,1,n}]]]
```

**Stack**[jf, acc] =
```
            Subst[jf UnitToken acc]
```

From Abelson and Sussman, *Structure and Interpretation of Computer Programs*

**18. abstractMachineFactorial** =  <p385>

**19. registerMachineFactorial** =  <p511>

**20. compiledFactorial** = <p596-7>