What is an Algorithm? Versions of Factorial

What is an algorithm? The text offers this: "An *algorithm* is a clearly specified set of instructions the computer will follow to solve a problem." This is essentially the same definition as given by one of the first books on algorithms: Aho, Hopcroft and Ullman, *Data Structures and Algorithms* (1983). The essential ingredients of an algorithm:

1. A sequence of steps
2. An unambiguous specification for each step
3. The steps can be carried out mechanically, ie by a machine.
4. Finite and terminating.

The above definition is bias toward a procedural model of computation. As well, it is blind to compiler and machine code transformations. What, for instance, does "clearly" mean? An algorithm fits the illusion that the user has about the computational process? An algorithm has a unique sequence of machine instructions? An algorithm is a mathematical construct divorced from implementation details?

Why a sequence of steps? Are parallel, distributed, and concurrent algorithms not really algorithms? Recall that *all* silicon processes occur in parallel.

Why unambiguous? A proof, for example, can take many different forms, using completely different steps and theories, but still arriving at the same result. Are non-deterministic, learning, chaotic, quantum, and other modern computational techniques not using algorithms?

Why mechanical? This one is easy: because Computer Science studies machines.

Why finite? When you write a control loop that waits for input, it is an algorithm?

Are algorithms different than data structures? Consider what some founders of Computer Science (Wirth, Djkstra, Hoare) said thirty years ago: "Decisions about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often strongly depend on the structure of the underlying data. In short, the subjects of program composition and data structures are inseperably intertwined." (in Wirth, Algorithms + Data Structures = Programs (1976), pxii)

Why then do object-oriented, logical, functional, and mathematical text books not mention algorithms at all? How does *algorithm analysis* work if it does not include notions of data structure or underlying engine or application context?

Algorithm analysis assumes that algorithms are mathematical procedures which have clearly definable upper limits on resource usage. The upper limits on algorithmic complexity provide little useful information about processes; algorithm analysis primarily tells us whether or not an encoding is tractable or intractable.

Tractable algorithms do not require exponentially increasing resources as problem size increases. Finding out whether or not an arbitrary Boolean expression has any solutions (the *satisifability* problem) is thought to be an intractable problem. And this is the simplest of all non-trivial mathematical questions in the simplest of all non-trivial mathematical domains. Recall that Boolean computation is all that occurs at the silicon level of computation. Why can algorithm analysis not even tell us if satisfiability is tractable or intractable?

Examine each of these algorithms for computing the Factorial function. Which is best? Which is most efficient (this is a trick question, why?)? Which are algorithms?

```
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


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


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


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


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] } ]


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)


recursiveFactorial[n] :=
      if[  n == 1, 1,  n*recursiveFactorial[n - 1] ]
```

```
rulebasedFactorial[1] = 1;
rulebasedFactorial[n] := n * rulebasedFactorial[n - 1]


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


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


mathematicalFactorial[n] =
      Apply[ Times, Range[n] ]


generatorFactorial[n]
      Times[ i, Generator[i, 1, n] ]


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


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


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


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

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

```
abstractMachineFactorial =  <p385>

registerMachineFactorial =  <p511>

compiledFactorial = <p596-7>
```