CSci 555: Functional Programming
Recursion Styles, Correctness, and Efficiency
— Scala Version —

H. Conrad Cunningham

7 February 2019

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of February 2019 is a recent version of Firefox from Mozilla.

Recursion Styles, Correctness, and Efficiency

Introduction

This set of notes introduces basic recursive programming styles and examines issues of termination, correctness, and efficiency.

The goals of the chapter are to:

- explore several recursive programming styles—linear and nonlinear, backward and forward, tail, and logarithmic—and their implementation using Scala
- examine how to analyze Scala functions to determine under what conditions they terminate with the correct result and how efficient they are
- explore methods for developing recursive Scala programs that terminate with the correct result and are efficient in both time and space usage

Note: The source code for the functions in these notes are in the Scala file `RecursionStyles.scala`.

Linear and Nonlinear Recursion

In this section, we examine the concepts of linear and nonlinear recursion. The following two sections examine other styles.

Linear recursion

A function definition is *linear recursive* if at most one recursive application of the function occurs in a leg of the definition (i.e. along a path from an entry to a return). The various function clauses and branches of conditional expressions (e.g. `if` and `match`) introduce paths.

The definition of the function `factorial` below is linear recursive because the expression in the second leg of the definition (i.e. `n * factorial(n-1)`) involves a single recursive application. The other leg is nonrecursive; it is the base case of the recursive definition.

```
def factorial(n: Int): Int = n match {
    case 0          => 1
    case m if m > 0 => m * factorial(m-1)  // linear rec.
    case _          =>
        sys.error(s"Factorial undefined for $n")
}
```

Scala checks for pattern matches for the clauses in the order given in the function definition. It executes the leg corresponding to the first successful match. If no pattern matches, then the function aborts and displays an error message.

Termination of recursion

How do we know that function `factorial` terminates?

To show that evaluation of a recursive function terminates, we must show that each recursive application *always* gets closer to a normal termination condition represented by a base case.

For a call `factorial(n)` with `n > 0`, the argument of the recursive application always decreases to `n - 1`. Because the argument always decreases in integer steps, it must eventually reach 0 and, hence, terminate in the first leg of the definition.

Preconditions and postconditions

The *precondition* of a function is what the caller (i.e. the client of the function) must ensure holds when calling the function. A precondition may specify the valid combinations of values of the arguments. It may also record any constraints on the values of "global" data structures that the function accesses or modifies. (By "global" we mean any entity that is not a parameter or local variable of the function.)

If the precondition holds, the supplier (i.e. developer) of the function must ensure that the function terminates with the *postcondition* satisfied. That is, the function returns the required values and/or alters the "global" data structures in the required manner.

The precondition of the `factorial` function requires that argument `n` be a nonnegative integer value. We could use Scala's predefined `requires` method to ensure this precondition holds, but, in this version, if all pattern matches fail, the function call aborts with a standard error message.

The postcondition of `factorial` is that the result returned is the correct mathematical value of `n` factorial. The function `factorial` neither accesses nor modifies any global data structures.

Time and space complexity

How efficient is function `factorial`?

Function `factorial` recurses to a depth of `n`. It thus has *time complexity* O(`n`), if we count either the recursive calls or the multiplication at each level.

The *space complexity* is also O(`n`) because a new runtime stack frame is needed for each recursive call.

Nonlinear recursion

A *nonlinear recursion* is a recursive function in which the evaluation of some leg requires more than one recursive application.

For example, the naive Fibonacci number function `fib` shown below has two recursive applications in its third leg. When we apply this function to a non-negative integer argument greater than 1, we generate a pattern of recursive applications that has the "shape" of a binary tree. Some call this a *tree recursion*.

```
def fib(n: Int): Int = n match {
    case 0           => 0
    case 1           => 1
    case m if m >= 2 => fib(m-1) + fib(m-2) // double rec.
    case _           =>
        sys.error(s"Fibonacci undefined for $n")
}
```

What are the precondition and postcondition for `fib(n)`?

For `fib(n)`, the precondition `n >= 0` ensures that the function is defined. When called with the precondition satisfied, the postcondition is:

`fib(n)` = *Fibonacci*(`n`)

How do we know that `fib` terminates?

For the recursive case n >= 2, the two recursive calls have arguments that are 1 or 2 less than `n`. Thus every call gets closer to one of the two base cases.

What are the time and space complexities of function `fib`?

Function `fib` is combinatorially explosive, having a time complexity O(`fib(n)`).

The space complexity is O(`n`) because a new runtime stack frame is needed for each recursive call and the calls recurse to a depth of `n`.

An advantage of a linear recursion over a nonlinear one is that a linear recursion can be compiled into a *loop* in a straightforward manner. Converting a nonlinear recursion to a loop is, in general, difficult.

Backward and Forward Recursion

In this section, we examine the concepts of backward and forward recursion.

Backward recursion

A function definition is *backward recursive* if the recursive application is *embedded within another expression.* During execution, the program must complete the evaluation of the expression after the recursive call returns. Thus, the program must preserve sufficient information from the outer call's environment to complete the evaluation.

The definition for the function `factorial` above is backward recursive because the recursive application `factorial(n-1)` in the second leg is embedded within the expression `n * factorial(n-1)`. During execution, the multiplication must be done after return. The program must "remember" (at least) the value of parameter `n` for that call.

A compiler can translate a backward linear recursion into a loop, but the translation may require the use of a stack to store the program's *state* (i.e. the values of the variables and execution location) needed to complete the evaluation of the expression.

Often when we design an algorithm, the first functions we come up with are backward recursive. They often correspond directly to a convenient recurrence relation. It is often useful to convert the function into an equivalent one that evaluates more efficiently.

Forward recursion

A function definition is *forward recursive* if the recursive application is *not embedded within another expression.* That is, the *outermost expression is the recursive application* and any other subexpressions appear in the argument lists. During execution, significant work is done as the recursive calls are made (e.g. in the argument list of the recursive call).

The definition for the auxiliary function `factIter` within the `factorial2` definition below is forward recursive. The recursive application `factIter(m-1,m*r)` in the second leg is on the outside of the expression evaluated for return. The other legs are nonrecursive.

```
def factorial2(n: Int): Int = {

    def factIter(n: Int, r: Int): Int = n match {
        case 0          => r
        case m if m > 0 => factIter(m-1,m*r)
    }

    if (n >= 0)
        factIter(n,1)
    else
```

```
            sys.error(s"Factorial undefined for $n")
    }
```

What are the precondition and postcondition for `factIter(n,r)`?

To avoid termination, `factIter(n,r)` requires `n >= 0`. Its postcondition is that:

> `factIter(n,r) = r *` *fact*`(n)`

How do we know that `factIter` terminates?

Argument `n` of the recursive call is at least 1 and decreases by 1 on each recursive call; it eventually reaches the base case.

What is the time complexity of function `factorial2`?

Function `factIter(n,r)` has a time complexity of O(`n`). But, because, tail call optimization converts the `factIter` recursion to a loop, the time complexity's constant factor should be smaller than that of `factorial(n)`.

As shown, `factIter(n,r)` seems to have a space complexity of O(`n`). But tail call optimization converts the recursion to a loop. Thus the space complexity of `factIter(n,r)` becomes O(1).

Tail Recursion

A function definition is *tail recursive* if it is *both forward recursive and linear recursive*. In a tail recursion, the last action performed before the return is a recursive call.

The definition of the function `factIter` above is tail recursive because it is both forward recursive and linear recursive.

Tail recursive definitions are easy to compile into efficient loops. There is no need to save the states of unevaluated expressions for higher level calls; the result of a recursive call can be returned directly as the caller's result. This is sometimes called *tail call optimization* (or "tail call elimination" or "proper tail calls").

In converting the backward recursive function `factorial` to a tail recursive auxiliary function, we added the parameter `r` to `factIter`. This parameter is sometimes called an *accumulating parameter* (or just an *accumulator*).

We typically use an accumulating parameter to "accumulate" the result of the computation incrementally for return when the recursion terminates. In `factIter`, this "state" passed from one "iteration" to the next enables us to convert a backward recursive function to an "equivalent" tail recursive one.

Function `factIter(n,r)` defines a more general function than `factorial`. It computes a factorial when we initialize the accumulator to 1, but it can compute some multiple of the factorial if we initialize the accumulator to another value.

However, the application of `factIter` in `factorial2` gives the initial value of 1 needed for factorial.

Consider auxiliary function `fibIter` used by function `fib2` below. This function adds two “accumulating parameters” to the backward nonlinear recursive function `fib` to convert the nonlinear (tree) recursion into a tail recursion. This technique works for Fibonacci numbers, but the same technique will not work in all cases.

```
def fib2(n: Int): Int = {

    def fibIter(n: Int, p: Int, q: Int): Int = n match {
        case 0 => p
        case m => fibIter(m-1,q,p+q)
    }

    if (n >= 0)
        fibIter(n,0,1)
    else
        sys.error(s"Fibonacci undefined for $n")
}
```

What are the precondition and postcondition for `fibIter(n,p,q)`?

To avoid abnormal termination, `fibIter(n,p,q)` requires `n >= 0`. When the precondition holds, its postcondition is:

`fibIter(n,p,q) =` *Fibonacci*`(n) + (p + q - 1)`

How do we know that `fibIter` terminates?

The recursive leg of `fibIter(n,p,q)` is only evaluated when `n1 > 0`. On the recursive call, that argument decreases by 1. So eventually the computation reaches the base case.

What are the time and space complexities of `fibIter`?

Function `fibIter` has a time complexity of O(`n`) in contrast to O(`fib(n)`) for `fib`. This algorithmic speedup results from the replacement of the very expensive operation `fib(n-1) + fib(n-2)` at each level in `fib` by the inexpensive operation `p + q` (i.e. addition of two numbers) in `fib2`.

Without tail call optimization, `fibIter(n,p,q)` has space complexity of O(`n`). However, tail call optimization can convert the recursion to a loop, giving O(1) space complexity.

When combined with tail-call optimization, a tail recursive function may be more efficient than the equivalent backward recursive function. However, the backward recursive function is often easier to understand and to reason about.

Logarithmic Recursive

We can define the exponentiation operator ^ in terms of multiplication as follows for integers b and n >= 0:

$$\texttt{b\^{}n} = \prod_{i=1}^{i=n} b$$

The backward recursive exponentiation function expt1 below raises a number to a nonnegative integer power. It has time complexity O(n) and space complexity O(n).

```
def expt1(b: Double, n: Int): Double = n match {
    case 0          => 1
    case m if m > 0 => b * expt1(b,m-1)
    case _          =>
        sys.error(s"Cannot raise to a negative power $n")
}
```

Consider the following questions relative to expt1(b,n).

- What are the precondition and postcondition for expt1(b,n)?
- How do we know that expt1(b,n) terminates?
- What are the time and space complexities for expt1(b,n)?

We can define a tail recursive auxiliary function exptIter by adding a new parameter p to accumulate the value of the exponentiation incrementally. We can define exptIter within a function expt2, taking advantage of the fact that the base b does not change. This is shown below.

```
def expt2(b: Double, n: Int): Double = {

    def exptIter(n: Int, p: Double): Double =
        n match {
            case 0 => p
            case m => exptIter(m-1,b*p)
        }

    if (n >= 0)
        exptIter(n,1)
    else
        sys.error(s"Cannot raise to negative power $n")
}
```

Consider the following questions relative to expt1(b,n).

- What are the precondition and postcondition for exptIter(n,p)?
- How do we know that exptIter(n,p) terminates?
- What are the time and space complexities for exptIter(n,p)?

The exponentiation function can be made computationally more efficient by squaring the intermediate values instead of iteratively multiplying. We observe that:

```
b^n = b^(n/2)^2   if n is even
b^n = b * b^(n-1) if n is odd
```

Function `expt3` below incorporates this observation in an improved algorithm. Its time complexity is O(`log(n)`) and space complexity is O(`log(n)`).

```
def expt3(b: Double, n: Int): Double = {

    def exptAux(n: Int): Double = n match {
        case 0                     => 1
        case m if (m % 2 == 0) => // i.e. even
            val exp = exptAux(m/2)
            exp * exp              // backward recursion
        case m                     => // i.e. odd
            b * exptAux(m-1)       // backward recursion
    }

    if (n >= 0)
        exptAux(n)
    else
        sys.error(s"Cannot raise to negative power $n")
}
```

Consider the following questions relative to `expt3`.

- What are the precondition and postcondition of `expt3(b,n)`?
- How do we know that `exptAux(n)` terminates?
- What are the time and space complexities of `exptAux(n)`?

Exercises

TODO: I adapted many of these exercise descriptions from similar Haskell exercises in ELIFP [Cunningham 2018] Chapters 5 and 9. They should be reconsidered, refined, and tested better for use in a Scala-based functional programming course. The order may also need to be modified and some exercises are probably better placed with different notes.

1. Answer the questions (precondition, postcondition, termination, time complexity, space complexity) in the discussion of `expt1`.
2. Answer the questions in the discussion of `expt2` and `exptIter`.
3. Answer the questions in the discussion of `expt3` and `exptAux`.

4. Develop a Scala function `sumSqBig` that takes three `Double` arguments and returns the sum of the squares of the two larger numbers.

   For example, `sumSqBig(2.0,1.0,3.0)` yields `13.0`.

5. Develop a Scala function `prodSqSmall` that takes three `Double` arguments and returns the product of the squares of the two smaller numbers.

   For example, `prodSqSmall(2.0,4.0,3.0)` yields `36.0`.

6. Develop a Scala function `xor` that takes two Boolean arguments and returns the "exclusive-or" of the two values. An exclusive-or operation returns `true` when exactly one of its arguments is `true` and returns `false` otherwise.

7. Develop a Scala function `implies` that takes two Boolean arguments `p` and `q` and returns the Boolean result `p` ⇒ `q` (i.e. logical implication). That is, if `p` is `true` and `q` is `false`, then the result is `false`; otherwise, the result is `true`.

   Note: This function is sometimes called `nand`.

8. Develop a Scala function `div23n5` that takes an `Int` and returns the Boolean `true` if and only if the integer is divisible by 2 or divisible by 3, but is not divisible by 5.

   For example, `div23n5(4)`, `div23n5(6)`, and `div23n5(9)` yield `true` and `div23n5(5)`, `div23n5(7_`, `div23n5(10)`, `div23n5(15)`, `div23n5(30)` yield `false`.

9. Develop a Scala function `notDiv` such that `notDiv(n,d)` returns `true` if and only if integer `n` is not divisible by integer `d`.

   For example, `notDiv(10,5)` yields `false` and `notDiv(11,5)` yields `true`.

10. Develop a Scala function `ccArea` that takes the *diameters* of two concentric circles (i.e. with the same center point) as `Double` values and returns the area of the space between the circles. That is, compute the area of the larger circle minus the area of the smaller circle.

    For example, `ccArea(2.0,4.0)` yields approximately `9.42477796`.

11. Develop a Scala function `addTax` that takes two `Double` values such that `addTax(c,p)` returns `c` with a sales tax of `p` *percent* added. For example, `addTax(2.0,9.0)` returns `2.18`.

    Also develop a function `subTax` that is the inverse of `addTax`. That is, `subTax((addTax(c,p)), p)` yields `c`. For example, `subTax(2.18,9.0) = 2.0`.

12. Develop a backward recursive Scala function `sumTo` such that `sumTo(n)` computes the sum of the integers from 1 to `n` for `n > 0`.

13. Develop a Scala function `sumTo2` such that `sumTo2(n)` computes the sum of the integers from 1 to `n` for `n > 0`. Use a tail recursive auxilliary function.

14. Develop a backward recursive Scala function `sumFromTo` such that `sumFromTo(m,n)` computes the sum of the integers from `m` to `n` for `n >= m`.

15. Develop a Scala function `sumFromTo2` such that `sumFromTo2(m,n)` computes the sum of the integers from `m` to `n` `n >= m`. Use a tail recursive auxilliary function.

16. Suppose we have Scala functions `succ` (successor) and `pred` (predecessor) defined as follows:

    ```
    def succ(n: Int): Int = n + 1
    def pred(n: Int): Int = n - 1
    ```

    Develop a recursive Scala function `add` such that `add(m,n)` computes `m + n` for two integers `m` and `n`. Function `add` *cannot* use addition or subtraction operators but *can* use unary negation, comparisons between integers, and the `succ` and `pred` functions defined above.

17. Develop a recursive Scala function `mult` such that `mult(m,n)` computes `m * n` for two integers `m` and `n`. The function *cannot* use the multiplication (`*`) or division (`/`) operators but *can* use unary negation, comparisons between integers, and the `succ`, `pred`, and `add` function from the previous exercise.

18. Develop a recursive Scala function `acker` to compute Ackermann's function, which is a function A defined as follows for integers `m` and `n`:

$A(m,n)$	$=$	$n+1,$	if $m = 0$
$A(m,n)$	$=$	$A(m-1,1),$	if $m > 0$ and $n = 0$
$A(m,n)$	$=$	$A(m-1, A(m, m-1)),$	if $m > 0$ and $n > 0$

19. Develop a recursive Scala function `hailstone` to implement the following function:

$hailstone(n)$	$=$	$1,$	if $n = 1$
$hailstone(n)$	$=$	$hailstone(n/2),$	if $n > 1$, even n
$hailstone(n)$	$=$	$hailstone(3*n+1),$	if $n > 1$, odd n

    Note that an application of the `hailstone` function to the argument `3` would result in the following "sequence" of "calls" and would ultimately return the result `1`.

    ```
    hailstone(3)
      hailstone(10)
        hailstone(5)
          hailstone(16)
            hailstone(8)
    ```

```
hailstone(4)
  hailstone(2)
    hailstone(1)
```

What is the domain of the *hailstone* function? How do we know the function terminates?

20. Develop a Scala exponentiation function `expt4` that is similar to `expt3` but is tail recursive as well as logarithmic recursive.

21. Develop the following group of recursive Scala functions:

    - `test` such that `test(a,b,c)` is `true` if and only if `a <= b` and no integer is the range from `a` to `b` inclusive is divisible by `c`.
    - `prime` such that `prime(n)` is `true` if and only if `n` is a prime integer.
    - `nextPrime` such that `nextPrime(n)` returns the next prime integer greater than `n`

22. Develop a recursive Scala function `binom` to compute *binomial coefficients*. That is, `binom(n,k)` returns $\binom{n}{k}$ for integers `n >= 0` and `0 <= k <= n`.

23. The time of day can be represented in Scala by the definitions

    ```
    sealed trait APM
    case object AM extends APM
    case object PM extends APM
    case class Time12(hours: Int, minutes: Int, apm: APM)
    ```

    where `hours` and `minutes` are integers such that `1 <= hours <= 12` and `0 <= minutes <= 59`.

    Develop a Boolean Scala function `comesBefore` that takes two `Time12` objects and determines whether the first is an earlier time than the second.

    TODO: Perhaps modify the exercise above to use the `Ord` trait as in the exercise below.

24. A date on the *proleptic Gregorian calendar* (see note below) can be represented in Scala by the definition

    ```
    case class PGDate(year: Int, month: Int, day: Int)
    ```

    with the following constraints on *valid* objects:

    - `year` is any integer
    - `1 <= month <= 12`
    - `1 <= day <= days_in_month(year,month)`

    Here `days_in_month(year,month)` represents the number of days in the the given `month` (i.e. 28, 29, 30, or 31) for the given `year`. Remember that the number of days in February varies between regular and leap years.

For the items below, write your own Scala functions. Do not use a date library.

a. Extend class `PGdate` to implement trait `Ord` as defined below (and in the *Notes on Scala for Java Programmers*):

```
trait Ord {
    def < (that: Any): Boolean
    def <=(that: Any): Boolean =
        (this < that) || (this == that)
    def > (that: Any): Boolean = !(this <= that)
    def >=(that: Any): Boolean = !(this < that)
}
```

   If needed, redefine the method `equals`.

   The interpretation of `d1 < d2` is that `d1` is an earlier date than `d2`.

b. Redefine method `toString` appropriately for `PGDate`.

c. Develop a Scala function `validPGDate(d)` that takes a `PGDate` object `d` and returns `true` if and only if `d` satisfies the constraints given above.

   For example:

   - `validPGDate(PGDate(2019,2,1)) == true`
   - `validPGDate(PGDate(2016,2,29)) == true`
   - `validPGDate(PGDate(2017,2,29)) == false`
   - `validPGDate(PGDate(0,0,0)) == false`

   You may need to develop one or more other functions to implement the `validPGDate` function.

d. For any `PGDate` beginning with (i.e. `>=`) `PGDate(-4712,1,1)`, develop Scala functions:

   - `daysBetween(d1,d2)` that takes two valid `PGDate` objects `d1` and `d2` and returns the number of days between them. The difference value is positive if `d1 < d2` and negative if `d1 > d2`.

   - `addDays(d,days)` takes a `PGDate` object `d` and an integer number of days and returns a valid `PGDate` object that is offset by that number of days. A positive offset results in a later date.

Note: The Gregorian calendar [Wikipedia 2019] was introduced by Pope Gregory of the Roman Catholic Church in October 1582. It replaced the Julian calendar system, which had been instituted in the Roman Empire by Julius Caesar in 46 BC. The goal of the change was to align the calendar year with the astronomical year.

Some countries adopted the Gregorian calendar at that time. Other countries adopted it later. Some countries may never have adopted it officially.

However, the Gregorian calendar system became the common calendar used worldwide for most civil matters. The *proleptic Gregorian calendar* [Wikipedia 2019] extends the calendar backward in time from 1582. The year 1 BC becomes year 0, 2 BC becomes year -1, etc. The proleptic Gregorian calendar underlies the ISO 8601 standard used for dates and times in software systems [Wikipedia 2019].

Arithmetic on calendar dates is often done by converting a date to the Julian Day Number (JDN), doing the arithmetic on those values, and then converting back to the calendar date [Wikipedia 2019].

25. Develop a Scala function `roman` that takes an `Int`) in the range from 0 to 3999 (inclusive) and returns the corresponding Roman numeral as a string (using capital letters). The function should halt with an appropriate `sys.error` messages if the argument is below or above the range. Roman numbers use the following symbols and are combined by addition or subtraction of symbols.

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For the purposes of this exercise, we represent the Roman numeral for 0 as the empty string. The Roman numbers for integers 1-20 are `I`, `II`, `III`, `IV`, `V`, `VI`, `VII`, `VIII`, `IX`, `X`, `XI`, `XII`, `XIII`, `XIV`, `XV`, `XVI`, `XVII`, `XVII`, `XIX`, and `XX`. Integers 40, 90, 400, and 900 are `XL`, `XC`, `CD`, and `CM`.

26. Develop a Scala function

```
def minf(g: (Int => Int)): Int
```

that takes a function `g` and returns the smallest integer `m` such that `0 <= m <= 10000000` and `g(m) == 0`. It should throw a `sys.error` if there is no such integer.

Acknowledgements

I wrote the first version of these notes in Fall 2013 to accompany my lectures on recursion concepts and programming techniques for a Lua-based course. I adapted some aspects of my earlier notes on functional programming using Haskell [Cunningham 2014].

I adapted the factorial, Fibonacci number, and exponentiation functions from similar Scheme functions in the classic textbook SICP [Abelson 1996].

I subsequently adapted these notes for use in functional or multiparadigm programming classes using Elixir (Spring 2015), Scala (Spring 2016), and Haskell (Summer 2016) [Cunningham 2016].

In Summer 2016, I also incorporated the Haskell version in what is now Chapter 9 of my Haskell-based textbook *Exploring Languages using Interpreters and Functional Programming* (ELIFP) [Cunningham 2018].

In Spring 2019, I merged parts of ELIFP Chapter 9 and the earlier Scala version of the notes to create the current document. I also included some exercises from ELIFP Chapter 5.

References

[**Abelson 1996**]**:** Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs* (SICP), Second Edition, MIT Press, 1996.

[**Bird 1988**]**:** Richard Bird and Philip Wadler. *Introduction to Functional Programming*, First Edition, Prentice Hall, 1988.

[**Cunningham 2014**]**:** H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1993-2014.

[**Cunningham 2016**]**:** H. Conrad Cunningham. Recursion Concepts and Terminology, 2013-2016.

[**Cunningham 2018**]**:** H. Conrad Cunningham. Exploring Languages with Interpreters ‘and Functional Programming, 2018. Available at https://john.cs.olemiss.edu/~hcc/csci450/ELIFP/ExploringLanguages.html.

[**Wikipedia 2019**]**:** *Wikipedia*, articles on “Gregorian Calendar”, “Proleptic Gregorian Calendar”, “Julian Day”, and “ISO 8601”, accessed on 31 January 2019.

Terms and Concepts

Recursion styles (linear vs. nonlinear, backward vs. forward, tail, and logarithmic), correctness (precondition, postcondition, and termination), efficiency estimation (time and space complexity), transformations to improve efficiency (auxiliary function, accumulator).