CSci 450: Org. of Programming Languages
Evaluation and Efficiency

H. Conrad Cunningham

16 September 2017

Contents

Acknowledgements: I adapted and revised this chapter from my previous materials in Summer and Fall 2016.

- Evaluation of Functional Programs from parts of chapters 1 and 13 of my *Notes on Functional Programming with Haskell* and from my notes on *Recursion Concepts and Terminology* (Scala version)
- three sections on recursion from my notes on *Recursion Concepts and Terminology* (Scala, Elixir, and Lua versions)
- the Haskell factorial, Fibonacci number, and exponentiation functions from my previous examples in Haskell, Elixir, Scala, Lua, and Elm, which, in turn, were adapted from the Scheme programs in Abelson and Sussman's classic, Scheme-based textbook SICP [Abelson-Sussman 1996].

In 2017, I continue to develop this chapter.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of September 2017 is a recent version of Firefox from Mozilla.

TODO:

- Review discussion of termination, preconditions, and postconditions for consistency with later discussion of those topics
- Add Java implementations of the `expt3` exponentiation function
- Add new examples?
- Update and add new exercises

3 Evaluation and Efficiency

3.1 Chapter Introduction

This chapter introduces an evaluation model and basic recursive programming styles and techniques applicable to Haskell programs. As in the previous chapter (Basic Haskell Functional Programming), it focuses on use of first-order functions and primitive data types.

The goals of the chapter are for the students to be able to

- analyze Haskell functions to determine under what conditions they terminate normally and how efficient they are
- develop recursive Haskell programs that terminate and are efficient in both time and space usage

Upon successful completion of this chapter, students should be able to:

1. explain the substitution model

2. use the substitution model informally to analyze Haskell programs to determine under what conditions they terminate normally and abnormally
3. use the substitution model informally to analyze Haskell programs to determine the time and space complexities of their execution
4. compare different implementations of the same functionality for termination and efficiency
5. use the appropriate programming techniques to develop Haskell programs that terminate and execute efficiently
6. compare basic functional programming syntax and semantics of Haskell to that in other programming languages

The Haskell code for this chapter is in file `EvalEff.hs`.

3.2 Evaluation of Functional Programs

How can we evaluate (i.e., execute) an expression that "calls" a function like the `fact1` function in a previous chapter?

We do this by rewriting expressions using a substitution model, as we see below. This process depends upon a property of functional languages called referential transparency.

3.2.1 Referential transparency

Referential transparency is probably the most important property of modern functional programming languages.

Referential transparency means that, within some well-defined context (e.g., a function or module definition), a variable (or other symbol) *always* represents the *same value*.

Because a variable always has the same value, we can replace the variable in an expression by its value or vice versa. Similarly, if two subexpressions have equal values, we can replace one subexpression by the other. That is, "equals can be replaced by equals".

Pure functional programming languages thus use the same concept of a variable that mathematics uses.

However, in most imperative programming languages, a variable represents an address or "container" in which values may be stored. A program may change the value stored in a variable by executing an assignment statement. Thus these mutable variables break the property of referential transparency.

Because of referential transparency, we can construct, reason about, and manipulate functional programs in much the same way we can any other mathematical

expressions. Many of the familiar "laws" from high school algebra still hold; new laws can be defined and proved for less familiar primitives and even user-defined operators. This enables a relatively natural equational style of reasoning using the actual expressions of the language.

In contrast, to reason about imperative programs, we usually need to go outside the language itself and use notation that represents the semantics of the language.

For our purposes here, referential transparency underlies the substitution model for evaluation of expressions in functional programs.

3.2.2 Substitution model

The *substitution model* (or *reduction model*) involves rewriting (or reducing) an expression to a "simpler" equivalent form. It involves two kinds of replacements:

- replacing a subexpression that satisfies the left-hand side of an equation by the right-hand side with appropriate substitution of arguments for parameters
- replacing a primitive application (e.g., `+` or `*`) by its value

The term *redex* refers to a subexpression that can be reduced.

Redexes can be selected for reduction in several ways. For instance, the redex can be selected based on its position within the expression:

- *leftmost redex first*, where the leftmost reducible subexpression in the expression text is reduced before any other subexpressions are reduced
- *rightmost redex first*, where the rightmost reducible subexpression in the expression text is reduced before any other subexpressions are reduced

The redex can also be selected based on whether or not it is contained within another redex:

- *outermost redex first*, where a reducible subexpression that is not contained within any other reducible subexpression is reduced before one that is contained within another
- *innermost redex first*, where a reducible subexpression that contains no other reducible subexpression is reduced before one that contains others

We will explore these more fully later in these notes. In most circumstances, Haskell uses a *leftmost outermost redex first* approach.

In a previous chapter, we defined factorial function `fact1` as follows:

```
fact1 :: Int -> Int
fact1 n = if n == 0 then
              1
          else
```

```
n * fact1 (n-1)
```

Consider the expression from `else` clause in `fact1` with `n` having the value `2`:

```
2 * fact1 (2-1)
```

This has two redexes: subexpressions `2-1` and `fact1 (2-1)`.

The multiplication cannot be reduced because it requires both of its arguments to be evaluated.

A function parameter is said to be *strict* if the value of that argument is always required. Thus, multiplication is strict in both its arguments. If the value of an argument is not always required, then it is *nonstrict*.

The first redex `2-1` is an innermost redex. Since it is the only innermost redex, it is both leftmost and rightmost.

The second redex `fact1 (2-1)` is an outermost redex. Since it is the only outermost redex, it is both leftmost and rightmost.

Now consider the complete evaluation of the expression `fact1 2` using leftmost outermost reduction steps. Below we denote the steps with $\Longrightarrow$ and give the substitution performed between braces.

```
     fact1 2
⟹   { replace fact1 2 using definition }
     if 2 == 0 then 1 else 2 * fact1 (2-1)
⟹   { evaluate 2 == 0 in condition }
     if False then 1 else 2 * fact1 (2-1)
⟹   { evaluate if }
     2 * fact1 (2-1)
⟹   { replace fact1 (2-1) using definition, add implicit parentheses }
     2 * (if (2-1) == 0 then 1 else (2-1) * fact1 ((2-1)-1))
⟹   { evaluate 2-1 in condition }
     2 * (if 1 == 0 then 1 else (2-1) * fact1 ((2-1)-1))
⟹   { evaluate 1 == 0 in condition }
     2 * (if False then 1 else (2-1) * fact1 ((2-1)-1))
⟹   { evaluate if }
     2 * ((2-1) * fact1 ((2-1)-1))
⟹   { evaluate leftmost 2-1 }
     2 * (1 * fact1 ((2-1)-1))
⟹   { replace fact1 ((2-1)-1) using definition, add implicit parentheses }
     2 * (1 * (if ((2-1)-1) == 0 then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹   { evaluate 2-1 in condition }
     2 * (1 * (if (1-1) == 0 then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹   { evaluate 1-1 in condition }
     2 * (1 * (if 0 == 0 then 1
```

```
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹   { evaluate 0 == 0 }
     2 * (1 * (if True then 1
     else ((2-1)-1) * fact1 ((2-1)-1)-1))
⟹   { evaluate if }
     2 * (1 * 1)
⟹   { evaluate 1 * 1 }
     2 * 1
⟹   { evaluate 2 * 1 }
     2
```

The rewriting model we have been using so far can be called *string reduction* because our model involves the textual replacement of one string by an equivalent string.

A more efficient alternative is *graph reduction*. In this technique, the expressions are represented as (directed acyclic) expression graphs rather than text strings. The repeated subexpressions of an expression are represented as shared components of the expression graph. Once a shared component has been evaluated once, it need not be evaluated again.

In the example above, subexpression `2-1` is reduced three times. However, all of those subexpressions come from the initial replacement of `fact1 2`. Using graph reduction, only the first of those reductions is necessary.

```
     fact1 2
⟹   { replace fact1 2 using definition }
     if 2 == 0 then 1 else 2 * fact1 (2-1)
⟹   { evaluate 2 == 0 in condition }
     if False then 1 else 2 * fact1 (2-1) }
⟹   { evaluate if }
     2 * fact1 (2-1)
⟹   { replace fact1 (2-1) using definition, add implicit parentheses }
     2 * (if (2-1) == 0 then 1 else (2-1) * fact1 ((2-1)-1))
⟹   { evaluate 2-1 because of condition (3 occurrences in graph) }
     2 * (if 1 == 0 then 1 else 1 * fact1 (1-1))
⟹   { evaluate 1 == 0 }
     2 * (if False then 1 else 1 * fact1 (1-1))
⟹   { evaluate if }
     2 * (1 * fact1 (1-1))
⟹   { replace fact1 ((1-1) using definition, add implicit parentheses }
     2 * (1 * (if (1-1) == 0 then 1 else (1-1) * fact1 ((1-1)-1))
⟹   { evaluate 1-1 because of condition (3 occurrences in graph) }
     2 * (1 * (if 0 == 0 then 1 else 0 * fact1 (0-1))
⟹   { evaluate 0 == 0 }
     2 * (1 * (if True then 1 else  0 * fact1 (0-1))
```

```
⟹   { evaluate if }
     2 * (1 * 1)
⟹   { evaluate 1 * 1 }
     2 * 1
⟹   { evaluate 2 * 1 }
     2
```

In general, the Haskell compiler or interpreter uses a leftmost outermost graph reduction technique. However, if the value of a function's argument is always needed for a computation, then an innermost reduction can be triggered for that argument. Either the programmer can explicitly require this or the compiler can detect the situation and automatically trigger the innermost reduction order.

Haskell exhibits *lazy evaluation*. That is, an expression is not evaluated until its value is needed, if ever. An outermost reduction corresponds to this evaluation strategy.

Other functional languages such as Scala and F# exhibit *eager evaluation*. That is, an expression is evaluated as soon as possible. An innermost reduction corresponds to this evaluation strategy.

3.2.3 Time and space complexity

We state efficiency (i.e., time complexity or space complexity) of programs in terms of the "Big-O" notation and asymptotic analysis.

For example, consider the leftmost outermost graph reduction of function `fact1` above. The number of reduction steps required to evaluate `fact1 n` is `5n + 3`.

We let the number of steps in a graph reduction be our measure of time. Thus, the *time complexity* of `fact1 n` is O(`n`), which means that the time to evaluate `fact1 n` is bounded above by some (mathematical) function that is proportional to the value of `n`.

Of course, this result is easy to see in this case. The algorithm is dominated by the `n` multiplications it must carry out. Alternatively, we see that evaluation requires on the order of `n` recursive calls.

We let the number of *arguments* in an expression graph be our measure of the *size* of an expression. Then the *space complexity* is the maximum size needed for the evaluation in terms of the input.

This size measure is an indication of the maximum size of the unevaluated expression that is held at a particular point in the evaluation process. This is a bit different from the way we normally think of space complexity in imperative algorithms, that is, the number of "words" required to store the program's data.

However, this is not as strange as it may at first appear. As we see later in these notes, the data structures in functional languages like Haskell are themselves *expressions* built by applying constructors to simpler data.

In the case of the graph reduction of `fact1 n`, the size of the largest expression is `2n + 16`. This is a multiplication for each integer in the range from 1 to `n` plus 16 for the full `if` statement. Thus the space complexity is O(`n`).

The Big-O analysis is an asymptotic analysis. That is, it estimates the order of magnitude of the evaluation time or space as the size of the input approaches infinity (gets large). We often do worst case analyses of time and space. Such analyses are usually easier to do than average-case analyses.

The time complexity of `fact1 n` is similar to that of a loop in an imperative program. However, the space complexity of the imperative loop algorithm is O(1). So `fact1` is not space efficient compared to the imperative loop.

We examine techniques for improving the efficiency of functions below. Later in these notes, we examine reduction techniques more fully.

3.2.4 Termination

A recursive function has one or more recursive cases and one or more base (nonrecursive) cases. It may also be undefined for some cases.

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

Again consider `fact1` defined above.

If `fact1` is called with argument `n` greater than 0, the argument of the recursive application in the `else` clause 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.

If we call `fact1` with argument 0, the function terminates immediately.

What if we call `fact1` with its argument less than 0? We consider this issue below.

3.2.5 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 access or modifies.

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.

Functions `fact1`, `fact2`, and `fact3` require that argument `n` be a natural number (nonnegative integer) value. If they are applied to a negative value for `n`, then the evaluation does not terminate. Operationally, they go into an "infinite loop" and likely will terminate when the runtime stack overflows.

If function `fact4` is called with a negative argument, then all guards and pattern matches fail. Thus the function aborts with a standard error message.

Thus to ensure normal termination, we impose the precondition

```
n >= 0
```

on these factorial functions.

The postcondition of all five factorial functions is that the result returned is the correct mathematical value of `n` factorial. For `fact4`, that is:

`fact4 n =` *fact'(n)*

None of the five factorial functions access or modify any global data structures, so we do not include other items in the precondition or postcondition.

Function `fact5` is defined to be 1 for all arguments less than zero. So, if this is the desired result, we can weaken the precondition to allow all integer values, for example,

```
True
```

and strengthen the postcondition to give the results for negative arguments, for example:

`fact5 n = if n >= 0 then` *fact'*`(n) else 1`

In the discussion of data abstraction in a previous chapter, we introduced the concept of *invariant*. In that context, an invariant is a postcondition of all constructor functions in the public interface. It is both a precondition and postcondition of all functions that access or update an "object". If there destructor functions that explicitly release the resources of an an object, the invariant must be a precondition.

3.3 Linear and Nonlinear Recursion

Given the substitution model, we can now consider efficiency and termination in the design of recursive Haskell functions.

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

The Haskell code for this chapter is in file `EvalEff.hs`.

3.3.1 Linear recursion

A function definition is *linear recursive* if at most one recursive application of the function occurs in any leg of the definition (i.e., along any path from an entry to a return). The various argument patterns and guards and the branches of the conditional expression `if` introduce paths.

The definition of the function `fact4` repeated below is linear recursive because the expression in the second leg of the definition (i.e., `n * fact4 (n-1)`) involves a single recursive application. The other leg is nonrecursive; it is the base case of the recursive definition.

```
fact4 :: Int -> Int
fact4 n
    | n == 0 =  1
    | n >= 1 =  n * fact4 (n-1)
```

What are the precondition and postcondition for `fact4 n`?

As noted above, we must require a precondition of `n >= 0` to avoid abnormal termination. When the precondition holds, the result is:

`fact4 n` = *fact'(n)*

What are the time and space complexities of `fact4 n`?

Function `fact4` recurses to a depth of `n`. As we saw earlier for `fact1`, it 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.

How do we know that function `fact4 n` terminates?

For a call `fact4 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.

3.3.2 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 nonnegative 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*.

```
fib :: Int -> Int
fib 0          = 0
fib 1          = 1
fib n | n >= 2 = fib (n-1) + fib (n-2)
```

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

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

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

How do we know that `fib n` 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 `fib n`?

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.

3.4 Backward and Forward Recursion

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

3.4.1 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 `fact4` above is backward recursive because the recursive application `fact4 (n-1)` in the second leg is *embedded within the expression* `n * fact4 (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.

3.4.2 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` below has two integer arguments. The first argument is the number whose factorial is to be computed. The second argument accumulates the product incrementally as recursive calls are made.

The recursive application `factIter (n-1) (n*r)` in the second leg is on the outside of the expression evaluated for return. The other leg of `factIter` and `fact6` itself are nonrecursive.

```
fact6 :: Int -> Int
fact6 n = factIter n 1

factIter :: Int -> Int -> Int
factIter 0 r         = r
factIter n r | n > 0 = factIter (n-1) (n*r)
```

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 n r` terminates?

Argument `n` of the recursive leg is at least 1 and decreases by 1 on each recursive call.

What is the time and space complexity of `factIter n r`?

Function `factIter n r` has a time complexity O(`n`). But, if the compiler converts the `factIter` recursion to a loop, the time complexity's constant factor should be smaller than that of `fact4`.

As shown, `factIter n r` has space complexity of O(`n`). But, if the compiler does an innermost reduction on the second argument (because its value will always be needed), then the space complexity of `factIter` becomes O(1).

3.4.3 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 thus tail recursive.

Tail recursive definitions are relatively straightforward 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 `fact4` 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` defines a more general function than `fact4`. 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 `fact6` 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.

```
fib2 :: Int -> Int
fib2 n | n >= 0 = fibIter n 0 1
    where
        fibIter 0 p q         = p
        fibIter m p q | m > 0 = fibIter (m-1) q (p+q)
```

Here we use *type inference* for `fibIter`. Function `fibIter` could be declared

```
fibIter :: Int -> Int -> Int -> Int
```

but it was not necessary because Haskell can infer the type from the types involved in its defining expressions.

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

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

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

If called with `p` and `q` set to 0 and 1, respectively, then `fibIter` returns:

Fibonacci`(n)`

How do we know that `fibIter n p q` terminates?

The recursive leg of `fibIter n p q` is only evaluated when `n > 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 `fibIter`.

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

When combined with tail-call optimization and innermost reduction of strict arguments, 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, as we see in a later chapter, to reason about.

3.5 Logarithmic Recursion

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

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

In this subsection, we develop Haskell functions for exponentiation, but we also briefly examine how to code similar recursive functions in other programming languages.

3.5.1 Haskell

A backward recursive exponentiation function `expt`, shown below in Haskell, raises a number to a nonnegative integer power.

```
expt :: Integer -> Integer -> Integer
expt b 0        = 1
expt b n
    | n > 0     = b * expt b (n-1)  -- backward rec
    | otherwise = error (
```

```
                        "expt not defined for negative exponent "
                        ++ show n )
```

Here we use the unbounded integer type `Integer` for the parameters and return value.

Note that the recursive call of `expt` does not change the value of the parameter `b`.

Consider the following questions relative to `expt`.

- What are the precondition and postcondition for `expt b n`?
- How do we know that `expt b n` terminates?
- What are the time and space complexities of `expt b n` (ignoring any additional costs of processing the unbounded integer type)?

We can define a tail recursive auxiliary function `exptIter` by adding a new parameter 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.

```
expt2 :: Integer -> Integer -> Integer
expt2 b n | n < 0 = error (
                      "expt2 not defined for negative exponent
"
                      ++ show n )
expt2 b n         = exptIter n 1
    where exptIter 0 p = p
          exptIter m p = exptIter (m-1) (b*p) -- tail rec
```

Consider the following questions relative to `expt2`.

- 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 of `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 into an improved algorithm. Its time complexity is O(`log(n)`) and space complexity is O(`log(n)`).

```
expt3 :: Integer -> Integer -> Integer
expt3 _ n | n < 0 = error (
                      "expt3 not defined for negative exponent "
                      ++ show n )
```

```
expt3 b n         = exptAux n
    where exptAux 0 = 1
          exptAux n
            | even n    = let exp = exptAux (n `div` 2) in
                              exp * exp      -- backward rec
            | otherwise = b * exptAux (n-1)  -- backward rec
```

Here we are use two features of Haskell we have not used in the previous examples.

- Boolean function `even` returns `True` if and only if its integer argument is an even number. Similarly, `odd` returns `True` when its argument is an odd number.
- The `let` clause introduces `exp` as a local definition within the expression following `in` keyword, that is, within `exp * exp`.

  The `let` feature allows us to introduce new definitions in a bottom-up manner–first defining a symbol and then using it.

Consider the following questions relative to `expt3`.

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

We express the functions in this chapter in Haskell, but they are adapted from the classic textbook *Structure and Interpretation of Computer Programs* (SICP) by Abelman and Sussman, which used Scheme.

Now let's look at what the `expt3` function looks like in Scheme and other languages.

3.5.2 Scheme

Below is the Scheme language program for exponentiation similar to to `expt3` (called `fast-expt` in SICP). Scheme, a dialect of Lisp, is an impure, eagerly evaluated functional language with dynamic typing.

```
(define (expt3 b n)
  (cond
    ((< n 0) (error `expt3 "Called with negative exponent"))
    (else (expt_aux b n))))

(define (expt_aux b n)
  (cond
    ((= n 0) 1)
    ((even? n) (square (expt3 b (/ n 2))))
    (else (* b (expt3 b (- n 1))))))
```

```
(define (square x) (* x x))

(define (even? n) (= (remainder n 2) 0))
```

Scheme (and Lisp) represents both data and programs as s-expressions (nested list structures) enclosed in balanced parentheses. In the case of executable expressions, the first element of the list may be operator. For example, consider:

```
(define (square x) (* x x))
```

The `define` operator takes two arguments:

- a symbol being defined, in this case a function signature `(square x)` for a function named `square` with one formal parameter named `x`
- an expression defining the value of the symbol, in this case the expression `(* x x)` that multiplies formal parameter `x` by itself and returns the result

The `define` operator has the side effect of adding the definition of the symbol to the environment. That is, `square` is introduced as a one argument function with the value denoted by the expression `(* x x)`.

The conditional expression `cond` gives an if-then-elseif expression that evaluates a sequence of predicates until one evaluates to "true" value and then returns the paired expression. The `else` at the end always evaluates to "true".

The above Scheme code defines the functions `square`, the exponentiation function `expt3`, and the logical predicate `even?`. It uses the primitive Scheme functions `-`, `*`, `/`, `remainder`, and `=` (equality).

We can evaluate the Scheme expression `(expt 2 10)` using a Scheme interpreter (as I did using DrRacket) and get the value `1024`.

Although Haskell and Scheme are different in many ways – algebraic versus s-expression syntax, static versus dynamic typing, lazy versus eager evaluation (by default), always pure versus sometimes impure functions, etc. – the fundamental techniques we have examined in Haskell still apply to Scheme and other languages. We can use a substitution model, consider preconditions and termination, use tail recursion, and take advantage of first-class and higher-order functions.

Of course, each language offers a unique combination of features that can be exploited in our programs. For example, Scheme programmers can leverage its runtime flexibility and powerful macro system; Haskell programmers can build on its safe type system, algebraic data types, pattern matching, and other features.

Let's now consider other languages.

The Racket Scheme code for this subsection is in file `expt3.rkt`.

3.5.3 Elixir

The language Elixir is a relatively new language that executes on the Erlang platform (called the Erlang Virtual Machine or BEAM). Elixir is an eagerly evaluated functional language with strong support for message-passing concurrent programming. It is dynamically typed and is mostly pure except for input/output. It has pattern-matching features similar to Haskell.

We can render the `expt3` program into a sequential Elixir program as follows.

```
def expt3(b,n) when is_number(b) and is_integer(n)
                    and n >= 0 do
  exptAux(b,n)
end

defp exptAux(_,0) do 1 end

defp exptAux(b,n) do
  if rem(n,2) == 0  do  #  i.e. even
    exp = exptAux(b,div(n,2))
    exp * exp           #  backward rec
  else                  #  i.e. odd
    b * exptAux(b,n-1)  #  backward rec
  end
end
```

This code occurs within an Elixir module. The `def` statement defines a function that is exported from the module while `defp` defines a function that is private to the module (i.e., not exported).

A definition allows the addition of guard clauses following `when` (although they cannot include user-defined function calls because of restrictions of the Erlang VM). In function `expt3`, we use guards to do some type checking in this dynamically typed language and to ensure that the exponent is nonnegative.

Private function `exptAux` has two functions bodies. As in Haskell, the body is selected using pattern matching proceeding from top to bottom in the module. The first function body with the header `exptAux(_,0)` matches all cases in which the second argument is `0`. All other situations match the second header `exptAux(b,n)` binding parameters `b` and `n` to the argument values.

The functions `div` and `rem` denote integer division and remainder, respectively.

The Elixir `=` operator is not an assignment as in imperative languages. It is a pattern-match statement with an effect similar to `let` in the Haskell function.

Above the expression `exp = exptAux(b,div(n,2))` evaluates the recursive call and then binds the result to new local variable named `exp`. This value is used in the next statement to compute the return value `exp * exp`.

Again, although there are significant differences between Haskell and Elixir, the basic thinking and programming styles learned for Haskell are also useful in Elixir (or Erlang). These styles are also key to use of their concurrent programming features.

The Elixir code for this subsection is in file `expt.ex`.

3.5.4 Scala

The language Scala is a hybrid functional/object-oriented language that executes on the Java platform (i.e., on the Java Virtual Machine or JVM). Scala is an eagerly evaluated language. It allows functions to be written in a mostly pure manner, but it allows intermixing of functional, imperative, and object-oriented features. It has a relatively complex static type system similar to Java, but it supports type inference (although weaker than that of Haskell). It interoperates with Java and other languages on the JVM.

We can render the exponentiation function `expt3` into a functional Scala program as shown below. This uses the Java/Scala extended integer type `BigInt` for the base and return values.

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

  def exptAux(n1: Int): BigInt = // b known from outer
    n1 match {
      case 0                  => 1
      case m if (m % 2 == 0) =>  // i.e. even
        val exp = exptAux(m/2)
        exp * exp                // backward rec
      case m                  =>  // i.e. odd
        b * exptAux(m-1)         // backward rec
    }

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

The body of function `expt3` uses an `if-else` expression to ensure that the exponent is non-negative and then calls `exptAux` to do the work.

Function `expt3` encloses auxiliary function `exptAux`. For the latter, the parameters of `expt3` are in scope. For example, `exptAux` uses `b` from `expt3` as a constant.

Scala supports pattern matching using an explicit `match` operator in the form:

selector `match {` *alternatives* `}`

It evaluates the *selector* expression and then choses the first *alternative* pattern that matches this value, proceedings top to botton, left to right. We write the alternative as

`case` *pattern* `=>` *expression*

or with a guard as:

`case` *pattern* `if` *boolean_expression* `=>` *expression*

The *expression* may be a sequence of expressions. The value returned is the value of the last expression evaluated.

In this example, the `match` in `exptAux` could easily be replaced by an `if - else if - else` expression because it does not depend upon complex pattern matching.

In Haskell, functions are automatically curried. In Scala, we could alternatively define `expt3` in curried form using two argument lists as follows:

```
def expt3(b: BigInt)(n: Int): BigInt = ...
```

Again, we can use most of the functional programming methods we learn for Haskell in Scala. Scala has a few advantages over Haskell such as the ability to program in a multiparadigm style and interoperate with Java. However, Scala tends to be more complex and verbose than Haskell. Some features such as type inference and tail recursion are limited by Scala's need to operate on the JVM.

The Scala code for this subsection is in file `exptBigInt2.scala`.

3.5.5 Lua

Lua is a minimalistic, dynamically typed, imperative language designed to be embedded as a scripting language in other programs, such as computer games. It interoperates well with standard C and C++ programs.

We can render the exponentiation function `expt3` into a functional Lua program as shown below.

```
local function expt3(b,n)

  local function expt_aux(n)   -- b known from outer
    if n == 0 then
      return 1
    elseif n % 2 == 0 then     -- i.e. even
      local exp = expt_aux(n/2)
      return exp * exp         -- backward recursion
    else                       -- i.e. odd
      return b * expt_aux(n-1) -- backward recursion
    end
```

```
  end

  if type(b) == "number" and type(n) == "number" and n >= 0 and
      n == math.floor(n) then
    return expt_aux(n,1)
  else
    error("Invalid arguments to expt: " ..
          tostring(b) .. "^" .. tostring(n))
  end
end
```

Like the Scala version, we define the auxiliary function `expt_aux` inside of function `expt3`, limiting its scope to the outer function.

This function uses with Lua version 5.2. In this and earlier versions, the only numbers are IEEE standard floating point. As in the Elixir version, we make sure the arguments are numbers with the exponent argument being nonnegative. Given that the numbers are floating point, the function also ensures that the exponent is an integer.

Auxiliary function `expt_aux` does the computational work. It differentiates among the three cases using an `if` - `elseif` - `else` structure. Lua does not have a switch statement or pattern matching capability.

Lua is not normally considered a functional language, but it has a number of features that support functional programming – in particular, first-class and higher order functions and tail call optimization.

In many ways, Lua is semantically similar to Scheme, but instead of having the Lisp-like hierarchical list as its central data structure, Lua provides an efficient, mutable, associative data structure called a table (somewhat like a hash table or map in other languages). Lua does not support Scheme-style macros in the standard language.

Unlike Haskell, Elixir, and Scala, Lua does not have builtin immutable data structures or pattern matching. Lua programs tend to be relatively verbose. So some of the usual programming idioms from functional languages do not fit Lua well.

The Lua code for this subsection is in file `expt.lua`.

3.5.6 Elm

Elm is a new functional language intended primarily for client-side Web programming. It is currently compiled into JavaScript, so some aspects are limited by the target execution environment. For example, Elm's basic types are those of JavaScript. So integers are actually implemented as floating point numbers.

Elm has a syntax and semantics that is similar to, but simpler than, Haskell. It has a Haskell-like `let` construct for local definitions but not a `where` construct. It also limits pattern matching to structured types.

Below is an Elm implementation of an exponentiation function similar to the Haskell `expt3` function, except it is limited to the standard integers `Int`. Operator `//` denotes the integer division operation and `%` is remainder operator.

```
expt3 : Int -> Int -> Int
expt3 b n =
    let
        exptAux m =
            if m == 0 then
                1
            else if m % 2 == 0 then
                let
                    exp = exptAux (m // 2)
                in
                    exp * exp  -- backward rec
            else
                b  * exptAux (m-1)  -- backward rec
    in
        if n < 0 then
            0 -- error?
        else
            exptAux n
```

One semantic difference between Elm and Haskell is that Elm functions must be total–that is, return a result for every possible input. Thus, this simple function extends the definition of `expt3` to return 0 for a negative power. An alternative would be to have `expt3` return a `Maybe Int` type instead of `Int`. We will examine this feature in Haskell later.

The Elm code for this subsection is in file `expt.elm`.

3.6 Conclusion

As we have seen in this chapter, we can develop efficient programs using functional programming and the Haskell language. These may require use to think about problems and programming a bit differently than we might in an imperative or object-oriented language. However, the techniques we learn for Haskell are usually applicable whenever we use the functional paradigm in any language. The functional way of thinking can also improve our programming in more traditional imperative and object-oriented languages.

3.7 Exercises

1. Show the reduction of the expression `fib 4` using the substitution model.
2. Show the reduction of the expression `expt 4 3` using the substitution model.
3. Answer the questions (precondition, postcondition, termination, time complexity, space complexity) in the subsection about `expt`.
4. Answer the questions in the subsection about `exptAux`
5. Answer the questions in the subsection about `expt`.
6. Develop a recursive function in Java, C#, Python, or C++ that has the same functionality as `expt3`.

For each of the following exercises, develop a Haskell program. For each function, informally argue that it terminates and give Big-O time and space complexities. Also identify any preconditions necessary to guarantee correct operation. Take care that special cases and error conditions are handled in a reasonable way.

7. Develop a backward recursive function `sumTo` such that `sumTo n` computes the sum of the integers from 1 to `n` for `n >= 0`.
8. Develop a tail recursive function `sumTo'` such that `sumTo' n` computes the sum of the integers from 1 to `n` for `n >= 0`.
9. Develop a backward recursive function `sumFromTo` such that `sumFromTo m n` computs the sum of the integers from `m` to `n` for `m <= n`.
10. Develop a tail recursive function `sumFromTo'` such that `sumFromTo' m n` computes the sum of the integers from `m` to `n` for `m <= n`.
11. Suppose we have functions `succ` (successor) and `pred` (predecessor) defined as follows:

    ```
    succ, pred :: Int -> Int
    succ n = n + 1
    pred n = n - 1
    ```

    Develop a function `add` such that `add m n` computes `m + n`. Function `add` cannot use the integer addition or subtraction operations but can use the `succ` ad `pred` functions above.
12. Develop a function `acker` to compute Ackermann's function, which is function A defined as follows:

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

13. Develop a 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
```

    For further thought: What is the domain of the *hailstone* function?

14. Develop the exponentiation function `expt4` that is similar to `expt3` but is tail recursive.

15. Develop the following group of 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`

16. Develop function `binom` to compute *binomial coefficients*. That is, `binom n k` returns $\binom{n}{k}$ for integers `n >= 0` and `0 <= k <= n`.

TODO: Add more exercises for the techniques and features introduced in this section.

3.8 References

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

[Bird-Wadler 1998] Richard Bird and Philip Wadler. *Introduction to Functional Programming*, Second Edition, Addison Wesley, 1998. [First Edition, 1988]

[**Cunningham 2014**] H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1994-2014.

[**Thompson 2011**] Simon Thompon. *Haskell: The Craft of Programming, Third Edition*, Pearson, 2011.

3.9 Terms and Concepts

Referential transparency, redex, reduction strategies (leftmost vs. rightmost, innermost vs. outermost), string and graph reduction models, time and space complexity, termination. preconditions, postconditions, 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)