CSci 450: Org. of Programming Languages
List Programming

H. Conrad Cunningham

16 September 2017

Contents

Acknowledgements: In Summer 2016, I adapted and revised this chapter from chapter 5 of my *Notes on Functional Programming with Haskell* and from my notes on *Functional Data Structures* from the Spring 2016 Scala variant of CSci 555. The latter was based, in part, on chapter 3 of the book *Functional Programming in Scala* by Paul Chiusano and Runar Bjarnason (Manning, 2015).

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:

- Consider other examples–small and large
- Consider discussion of testing
- Consider integrating more discussion of abstract data types
- Mention other Prelude or library functions not discussed
- Edit and add more exercises (e.g., Candy Bowl, Cookie Jar)

4 List Programming

4.1 Chapter Introduction

This chapter introduces the list data type and develops the fundamental programming concepts and techniques for first-order polymorphic functions to process lists.

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

- develop correct Haskell functional programs consisting of first-order polymorphic functions that solve problems by processing lists

- develop Haskell list-processing programs that terminate and are efficient and elegant.

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

1. describe the basic syntax and semantics of first-order, polymorphic, Haskell functions for processing lists and strings

2. describe the Haskell list and string data types

3. develop first-order Haskell functional programs using lists, strings, pattern matching, polymorphism, and infix operators
4. apply the follow-types-to-implementations (form of algorithm follows the form of the data) and other techniques to develop Haskell programs
5. compare different implementations of the same functionality for termination and efficiency
6. use the appropriate programming techniques to develop Haskell list-processing programs that terminate and execute efficiently
7. appreciate the mathematical characteristics of Haskell functions (e.g., to state properties of functions and operators)

The Haskell module for this chapter is in `Mod04Lists.hs`.

4.2 Polymorphic List Data Type

As we have seen, to do functional programming, we construct programs from collections of pure functions. Given the same arguments, a *pure function* always returns the same result. The function application is thus referentially transparent.

Such a pure function does not have *side effects*. It does not modify a variable or a data structure in place. It does not throw an exception or perform input/output. It does nothing that can be seen from outside the function except return its value.

Thus the data structures in purely functional programs must be *immutable*, not subject to change as the program executes.

Functional programming languages often have a number of immutable data structures. However, the most salient one is the list.

We mentioned the Haskell list and string data types in a previous chapter. In this chapter, we look at lists in depth. Strings are just special cases of lists.

S### List: `[t]`

The primary built-in data structure in Haskell is the list, a sequence of values. All the elements in a list must have the same type. Thus we declare lists with the notation `[t]` to denote a list of zero or more elements of type `t`.

A *list* is is hierarchical data structure. It is either *empty* or it is a pair consisting of a *head element* and a *tail* that is itself a *list* of elements.

The Haskell list is an example of an *algebraic data type.* We discuss that concept in a later section.

A matching pair of empty square brackets (`[]`), pronounced “nil”, represents the empty list.

A colon (`:`), pronounced "cons", represents the *list constructor* operation between a *head* element on the left and a *tail* list on the right.

Example lists include:

```
[]
2:[]
3:(2:[])
```

The Haskell language adds a bit of *syntactic sugar* to make expressing lists easier. (By syntactic sugar, we mean notation that simplifies expression of a concept but that adds no new functionality to the language. The new notation can be defined in terms of other notation within the language.)

The cons operations binds from the right. Thus

```
5:(3:(2:[]))
```

can be written as:

```
5:3:2:[]
```

We can write this as a comma-separated sequence enclosed in brackets as follows:

```
[5,3,2]
```

Haskell supports two list selector functions, `head` and `tail`, such that

`head (h:t)` $\Longrightarrow$ `h`

where `h` is the head element of list, and

`tail (h:t)` $\Longrightarrow$ `t`

where `t` is the tail list.

Aside: Instead of `head`, Lisp uses `car` and other languages use `hd`, `first`, etc. Instead of `tail`, Lisp uses `cdr` and other languages use `tl`, `rest`, etc.

The Prelude library supports a number of other useful functions on lists. For example, `length` takes a list and returns its length.

Note that lists are defined inductively. That is, they are defined in terms of a base element `[]` and the list constructor operation cons (`:`). As you would expect, a form of mathematical induction can be used to prove that list-manipulating functions satisfy various properties. We will discuss in a later chapter.

4.2.1 String: `String`

In Haskell, a *string* is treated as a list of characters. Thus the data type `String` is defined as a *type synonym*:

```
type String = [Char]
```

In addition to the standard list syntax, a `String` literal can be given by a sequence of characters enclosed in double quotes. For example, `"oxford"` is shorthand for `['o','x','f','o','r','d']`.

Strings can contain any graphic character or any special character given as escape code sequence (using backslash). The special escape code `\&` is used to separate any character sequences that are otherwise ambiguous.

Example: `"Hello\nworld!\n"` is a string that has two newline characters embedded.

Example: `"\12\&3"` represents the list `['\12','3']`.

Because strings are represented as lists, all of the Prelude functions for manipulating lists also apply to strings.

Consider a function to compute the length of a string:

```
len :: String -> Int
len s  =  if s == [] then 0 else 1 + len (tail s)
```

Note that the argument string for the recursive application of `len` is simpler (i.e., shorter) than the original argument. Thus `len` will eventually be applied to a `[]` argument and, hence, `len`'s evaluation will terminate.

How efficient is this function (i.e., its time and space complexity)?

Consider the evaluation of the expression `len "five"`.

```
      len "five"
⟹    if "five" == [] then 0 else 1 + len (tail "five")
⟹    if False then 0 else 1 + len (tail "five")
⟹    1 + len (tail "five")
⟹    1 + len "ive"
⟹    1 + (if "ive" == [] then 0 else 1 + len (tail "ive"))
⟹    1 + (if False then 0 else 1 + len (tail "ive"))
⟹    1 + (1 + len (tail "ive"))
⟹    1 + (1 + len "ve")
⟹    1 + (1 + (if "ve" == [] then 0 else 1 + len (tail "ve")))
⟹    1 + (1 + (if False then 0 else 1 + len (tail "ve")))
⟹    1 + (1 + (1 + len (tail "ve")))
⟹    1 + (1 + (1 + len "e"))
⟹    1 + (1 + (1 + (if "e" == [] then 0 else 1 + len (tail "e"))))
⟹    1 + (1 + (1 + (if False then 0 else 1 + len (tail "e"))))
⟹    1 + (1 + (1 + (1 + len (tail "e"))))
⟹    1 + (1 + (1 + (1 + len "")))
⟹    1 + (1 + (1 + (1 + 0)))
⟹    1 + (1 + (1 + 1))
⟹    1 + (1 + 2)
⟹    1 + 3
```

The number of reduction steps in proportional to the length of the input list. Similarly, the amount of space required (given lazy evaluation) is also proportional to the length of the input list.

4.2.2 Polymorphic lists

The above definition of `len` only works for strings. How can we make it work for a list of integers or other elements?

For an arbitrary type `a`, we want `len` to take objects of type `[a]` and return an `Int` value. Thus its type signature could be:

```
len :: [a] -> Int
```

If `a` is a variable name (i.e., it begins with a lowercase letter) that does not already have a value, then the type expression `a` used as above is a *type variable*; it can represent an arbitrary type. All occurrences of a type variable appearing in a type signature must, of course, represent the same type.

An object whose type includes one or more type variables can be thought of having many different types and is thus described as having a *polymorphic type*. (The next subsection gives more detail on polymorphism in general.)

Polymorphism and first-class functions are powerful abstraction mechanisms: they allow irrelevant detail to be hidden.

Other examples of polymorphic list functions from the Prelude library include:

```
head :: [a] -> a
tail :: [a] -> [a]
(:)  :: a -> [a] -> [a]
```

4.2.3 Kinds of polymorphism

Polymorphism refers to the property of having "many shapes". In programming languages, we are primarily interested in how *polymorphic* function names (and operator symbols) are associated with implementations of the functions.

In general, two primary kinds of polymorphism exist in programming languages:

1. *Ad hoc polymorphism*, in which the same function name (or operator symbol) can denote different implementations depending upon how it is used in an expression. That is, the implementation invoked depends upon the types of function's arguments and return value.

   There are two subkinds of ad hoc polymorphism.

   a. *Overloading* refers to ad hoc polymorphism in which the language's compiler or interpreter determines the appropriate implementation to invoke using information from the context. In statically typed languages, overloaded names and symbols can usually be bound to the intended implementation at compile time based on the declared types of the entities. They exhibit *early binding*.

      Consider the language Java. It overloads a few operator symbols, such as using the `+` symbol for both addition of numbers and concatenation of strings. Java also overloads calls of functions defined with the same name but different signatures (patterns of parameter types and return value). Java does not support user-defined operator overloading; C++ does.

      Haskell's *type class* mechanism, which we examine in a later chapter, implements overloading polymorphism in Haskell.

   b. *Subtyping* (also known as *subtype polymorphism* or *inclusion polymorphism*) refers to ad hoc polymorphism in which the appropriate implementation is determined by searching a hierarchy of types. The function may be defined in a supertype and redefined (overridden) in subtypes. Beginning with the actual types of the data involved, the program searches up the type hierarchy to find the appropriate implementation to invoke. This usually occurs at runtime, so this exhibits *late binding*.

      The object-oriented programming community often refers to inheritance-based subtype polymorphism as simply *polymorphism*. This the polymorphism associated with the class structure in Java.

      Haskell does not support subtyping. Its type classes do support *class extension*, which enables one class to inherit the properties of another. However, Haskell's classes are not types.

2. *Parametric polymorphism*, in which the same implementation can be used for many different types. In most cases, the function (or class) implementation is stated in terms of one or more type parameters. In statically typed languages, this binding can usually be done at compile time (i.e., exhibiting early binding).

   The object oriented programming (e.g., Java) community often calls this type of polymorphism *generics* or *generic programming*.

   The functional programming (e.g., Haskell) community often calls this simply *polymorphism*.

4.3 Programming with List Patterns

In the factorial examples in an earlier chapter, we used integer patterns and guards to break out various cases of a function definition into separate equations. Lists and other data types may also be used in patterns.

Pattern matching helps enable the *form of the algorithm* to match the *form of the data structure*. Or, as others may say, it helps in *following types to implementations*.

This is considered elegant. It is also pragmatic. The structure of the data often suggests the algorithm that is needed for a task.

In general, lists have two cases that need to be handled: the empty list and the nonempty list. Breaking a definition for a list-processing function into these two cases is usually a good place to begin.

4.3.1 Summing a list of integers: `sum'`

Consider a function `sum'` to sum all the elements in a list of integers. That is, if the list is

$$v_1, v_2, v_3, \cdots, v_n,$$

then the sum of the list is the value resulting from inserting the addition operator between consecutive elements of the list:

$$v_1 + v_2 + v_3 + \cdots + v_n.$$

Because addition is an *associative* operation (that is, $(x + y) + z = x + (y + z)$ for any integers x, y, and z), the above additions can be computed in any order.

What is the sum of an empty list?

Because there are no numbers to add, then, intuitively, zero seems to be the proper value for the sum.

In general, if some binary operation is inserted between the elements of a list, then the result for an empty list is the *identity* element for the operation. Since $0 + x = x = x + 0$ for all integers x, zero is the identity element for addition.

Now, how can we compute the sum of a nonempty list?

Because a nonempty list has at least one element, we can remove one element and add it to the sum of the rest of the list. Note that the "rest of the list" is a simpler (i.e., shorter) list than the original list. This suggests a recursive definition.

The fact that Haskell defines lists recursively as a cons of a head element with a tail list suggests that we structure the algorithm around the structure of the *beginning* of the list.

Bringing together the two cases above, we can define the function `sum'` in Haskell as follows. This is similar to the Prelude function `sum`.

```
{- Function sum' sums a list of integers. It is similar to
   function sum in the Prelude.
-}
sum' :: [Int] -> Int
sum' []     = 0           -- nil list
sum' (x:xs) = x + sum' xs -- non-nil list
```

- As noted previously, all of the text between the symbol "`--`" and the end of the line represents a *comment*; it is ignored by the Haskell interpreter.

  The text enclosed by the `{-` and `-}` is a block comment, that can extend over multiple lines.

- This definition uses two *legs*. The equation in the first leg is used for nil list arguments, the second for non-nil arguments.

- Note the `(x:xs)` pattern in the second leg. The "`:`" denotes the list constructor operation *cons*.

  If this pattern succeeds, then the head element of the list argument is bound to the variable `x` and the tail of the list argument is bound to the variable `xs`. These bindings hold for the right-hand side of the equation.

- The use of the cons in the pattern simplifies the expression of the case. Otherwise the second leg would have to be stated using the `head` and `tail` selectors as follows:

  ```
  sum' xs = head xs + sum' (tail xs)
  ```

- We use the simple name `x` to represent items of some type and the name `xs`, the same name with an `s` (for sequence) appended, to represent a list of that same type. This is a useful convention (adopted from the classic Bird and Wadler textbook [Bird-Wadler 1998]) that helps make a definition easier to understand.

- Remember that patterns (and guards) are tested in the order of occurrence (i.e., left to right, top to bottom). Thus, in most situations, the cases should be listed from the most specific (e.g., nil) to the most general (e.g., non-nil).

- The length of a non-nil argument decreases by one for each successive recursive application. Thus `sum'` will eventually be applied to a `[]` argument and terminate.

For a list consisting of elements 2, 4, 6, and 8, that is, `2:4:6:8:[]`, function `sum'` computes

```
2 + (4 + (6 + (8 + 0)))
```

giving the integer result 20.

Function `sum'` is backward linear recursive; its time and space complexity are both $O(n)$, where n is the length of the input list.

We could, of course, redefine this to use a tail-recursive auxiliary function. With *tail call optimization*, the recursion could be converted into a loop. It would still be order $O(n)$ in time complexity (but with a smaller constant factor) but $O(1)$ in space.

4.3.2 Multiplying a list of numbers: `product'`

Now consider a function `product'` to multiply together a list of integers.

The product of an empty list is 1 (which is the identity element for multiplication).

The product of a nonempty list is the head of the list multiplied by the product of the tail of the list, except that, if a 0 occurs anywhere in the list, the product of the list is 0.

We can thus define `product'` with two base cases and one recursive case, as follows. This is similar to the Prelude function `product`.

```
product' :: [Integer] -> Integer
product' []     = 1
product' (0:_)  = 0
product' (x:xs) = x * product' xs
```

Note the use of the wildcard pattern underscore "`_`" in the second leg above. This represents a "don't care" value. In this pattern it matches the tail, but no value is bound; the right-hand side of the equation does not need the actual value.

0 is the *zero element* for the multiplication operation on integers. That is, for all integers x:

$$0 * x = x * 0 = 0$$

For a list consisting of elements 2, 4, 6, and 8, that is, `2:4:6:8:[]`, function `product'` computes:

```
2 * (4 * (6 * (8 * 1)))
```

which yields the integer result 384.

For a list consisting of elements 2, 0, 6, and 8, function `product'` "short circuits" the computation as:

```
2 * 0
```

Like `sum'`, function `product'` is backward linear recursive; it has a worst-case time complexity of $O(n)$, where n is the length of the input list. It terminates because the argument of each successive recursive call is one element shorter than the previous call, approaching the first base case.

As with `sum'`, we could redefine this to use a tail-recursive auxiliary function, which could evaluate in $O(1)$ space with tail call optimization.

Note that `sum'` and `product'` have similar computational patterns. In a later chapter, we look at how to capture the commonality in a single higher-order function.

4.3.3 Length of a list: `length'`

As another example, consider the function for the length of a list that we discussed earlier (as `len`). Using list patterns we can define `length’` as follows:

```
length' :: [a] -> Int
length' []     = 0               -- nil list
length' (_:xs) = 1 + length' xs -- non-nil list
```

Note the use of the wildcard pattern underscore “`_`”. In this pattern it matches the head, but no value is bound; the right-hand side of the equation does not need the actual value.

Given a finite list for its argument, does this function terminate? What are its time and space complexities?

This definition is similar to the definition for `length` in the Prelude.

4.3.4 Remove duplicate elements: `remdups`

Consider the problem of removing adjacent duplicate elements from a list. That is, we want to replace a group of adjacent elements having the same value by a single occurrence of that value.

As with the above functions, we let the form of the data guide the form of the algorithm, following the type to the implementation.

The notion of adjacency is only meaningful when there are two or more of something. Thus, in approaching this problem, there seem to be three cases to consider:

- The argument is a list whose first two elements are duplicates; in which case one of them should be removed from the result.
- The argument is a list whose first two elements are not duplicates; in which case both elements are needed in the result.

- The argument is a list with fewer than two elements; in which case the remaining element, if any, is needed in the result.

Of course, we must be careful that sequences of more than two duplicates are handled properly.

Our algorithm thus can examine the first two elements of the list. If they are equal, then the first is discarded and the process is repeated recursively on the list remaining. If they are not equal, then the first element is retained in the result and the process is repeated on the list remaining. In either case the remaining list is one element shorter than the original list. When the list has fewer than two elements, it is simply returned as the result.

If we restrict the function to lists of integers, we can define Haskell function `remdups` as follows:

```
remdups :: [Int] -> [Int]
remdups (x:y:xs)
  | x == y = remdups (y:xs)
  | x /= y = x : remdups (y:xs)
remdups xs = xs
```

- Note the use of the pattern `(x:y:xs)`. This pattern match succeeds if the argument list has at least two elements: the head element is bound to `x`, the second element to `y`, and the tail list to `xs`.
- Note the use of guards to distinguish between the cases where the two elements are equal (`==`) and where they are not equal (`/=`).
- In this definition the case patterns overlap, that is, a list with at least two elements satisfies both patterns. But since the cases are evaluated top to bottom, the list only matches the first pattern. Thus the second pattern just matches lists with fewer than two elements.

What if we wanted to make the list type polymorphic instead of just integers?

At first glance, it would seem to be sufficient to give `remdups` the polymorphic type `[a] -> [a]`. But the guards complicate the situation a bit.

Evaluation of the guards requires that Haskell be able to compare elements of the polymorphic type `a` for equality (`==`) and inequality (`/=`). For some types these comparisons may not be supported. (For example, suppose the elements are functions.) Thus we need to restrict the polymorphism to types in which the comparisons are supported.

We can restrict the range of types by using a *context* predicate. The following type signature restricts the polymorphism of type variable `a` to the built-in *type class* `Eq`, the group of types for which both equality (`==`) and inequality (`/=`) comparisons have been defined:

```
remdups :: Eq a => [a] -> [a]
```

Another useful context is the class `Ord`, which is an *extension* of class `Eq`. `Ord` denotes the class of objects for which the relational operators `<`, `<=`, `>`, and `>=` have been defined in addition to `==` and `/=`.

In most situations the type signature can be left off the declaration of a function. Haskell then attempts to infer an appropriate type. For `remdups`, the type inference mechanism would assign the type `Eq [a] => [a] -> [a]` . However, in general, it is good practice to give explicit type signatures.

Like the previous functions, `remdups` is backward linear recursive; it takes a number of steps that is proportional to the length of the list. This function has a recursive call on both the duplicate and non-duplicate legs. Each of these recursive calls uses a list that is shorter than the previous call, thus moving closer to the base case.

4.3.5 More list patterns

The following table shows Haskell parameter patterns, corresponding arguments, and the result of the attempted match.

Pattern	**Argument**	**Succeeds?**	**Bindings**
`1`	`1`	yes	none
`x`	`1`	yes	`x` ← `1`
`(x:y)`	`[1,2]`	yes	`x` ← `1`, `y` ← `[2]`
`(x:y)`	`[[1,2]]`	yes	`x` ← `[1,2]`, `y` ← `[]`
`(x:y)`	`["olemiss"]`	yes	`x` ← `"olemiss"`, `y` ← `[]`
`(x:y)`	`"olemiss"`	yes	`x` ← `'o'`, `y` ← `"lemiss"`
`(1:x)`	`[1,2]`	yes	`x` ← `[2]`
`(1:x)`	`[2,2]`	no	none
`(x:_:_:y)`	`[1,2,3,4,5,6]`	yes	`x` ← `1`, `y` ← `[4,5,6]`
`[]`	`[]`	yes	none
`[x]`	`["Cy"]`	yes	`x` ← `"Cy"`
`[1,x]`	`[1,2]`	yes	`x` ← `2`
`[x,y]`	`[1]`	no	none
`(x,y)`	`(1,2)`	yes	`x` ← `1`, `y` ← `2`

4.4 Data sharing

Suppose we have the declaration:

```
xs = [1,2,3]
```

As we learned in the data structures course, we can implement this list as a singly linked list `xs` with three cells with the values `1`, `2`, and `3`, as shown in the figure below.

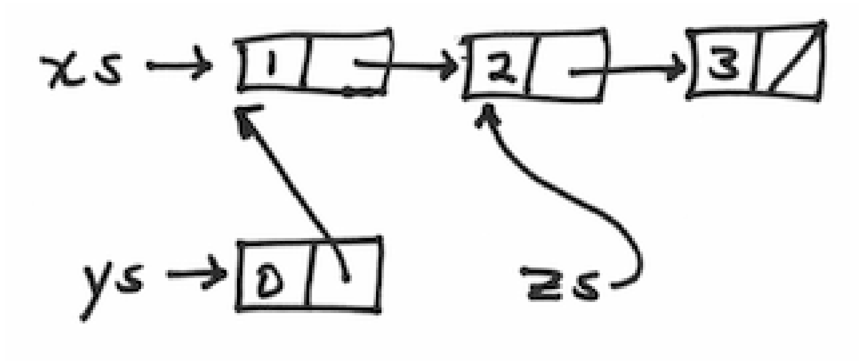

Figure 4-1: Data sharing in lists

Consider the following declarations

```
ys = 0:xs
zs = tail xs
```

where

- `0:xs` returns a list that has a new cell containing `0` in front of the previous list
- `tail xs` returns the list consisting of the last two elements of `xs`

If the linked list `xs` is immutable (i.e., the values and pointers in the three cells cannot be changed), then neither of these operations requires any copying.

- The first just constructs a new cell containing `0`, links it to the first cell in list `xs`, and initializes `ys` with a reference to the new cell.
- The second just returns a reference to the second cell in list `xs` and initializes `zs` with this reference.
- The original list `xs` is still available, unaltered.

This is called *data sharing*. It enables the programming language to implement immutable data structures efficiently, without copying in many key cases.

Also, such functional data structures are *persistent* because existing references are never changed by operations on the data structure.

Consider evaluation of the expression `head xs`. It must create a copy of the head element (in this case `1`). The result does not share data with the input list.

Similarly, the list returned by function `remdups` (defined above) does not share data with its input list.

4.4.1 Preconditions for `head` and `tail`

What should `tail` return if the list is nil?

One choice is to return a nil list `[]`. However, it seems illogical for an empty list to have a tail.

Consider a typical usage of the `tail` function. It is normally an error for a program to attempt to get the tail of an empty list. Moreover, a program can efficiently check whether a list is empty or not. So, in this case, it is better to consider `tail` a partial function.

Thus, Haskell defines both `tail` and `head` to have the precondition that their parameters are non-nil lists. If we call either with a nil list, then it will terminate execution with a standard error message.

4.4.2 Dropping elements from beginning of list

We can generalize `tail` to a function `drop'` that removes the first `n` elements of a list as follows, (This function is called `drop` in the Prelude.)

```
drop' :: Int -> [a] -> [a]
drop' n xs | n <= 0 = xs
drop' _ []          = []
drop' n (_:xs)      = drop' (n-1) xs
```

Example:

`drop 2 "oxford"` $\Longrightarrow \cdots$ `"ford"`

This function takes a different approach to the "empty list" issue than `tail` does. Although it is illogical to take the `tail` of an empty list, dropping the first element from an empty list seems subtly different. Given that we often use `drop'` in cases where the length of the input list is unknown, dropping the first element of an empty list does not necessarily indicate a program error.

Suppose instead that `drop'` would trigger an error when called with an empty list. To avoid this situation, the program might need to determine the length of the list argument. This is inefficient, usually requiring a traversal of the entire list to count the elements. Thus the choice for `drop'` to return a nil is also pragmatic.

The `drop'` function is tail recursive. The result list shares space with the input list.

The `drop'` function terminates when either the list argument is empty or the integer argument is 0 or negative. The function eventually terminates because each recursive call both shortens the list and decrements the integer.

What is the time complexity of `drop'`?

There are two base cases. For the first leg, the function must terminate in $O(max(1,$`n`$))$ steps. For the second leg, the function must terminate within $O($`length xs`$)$ steps. So the function must terminate within $O(min(max(1,$`n`$),$`length xs`$))$ steps.

What is the space complexity of `drop'`?

This tail recursive function evaluates in constant space when optimized.

4.4.3 Taking elements from the beginning of a list

Similarly, we can generalize `head'` to a function `take` that takes a number `n` and a list and returns the first `n` elements of the list.

```
take' :: Int -> [a] -> [a]
take' n _ | n <= 0 = []
take' _ []         = []
take' n (x:xs)     = x : take' (n-1) xs
```

Consider the following questions for this function?

- What is returned when the list argument is nil?
- Does evaluation of this function terminate?
- Does the result share data with the input?
- Is the function tail recursive?
- What are its time and space complexities?

Example:

`take 2 "oxford"` $\Longrightarrow \cdots$ `"ox"`

4.5 Using Infix Operations

In Haskell, a *binary operation* is a function of type `t1 -> t2 -> t3` for some types `t1`, `t2`, and `t3`.

We usually prefer to use *infix* syntax rather than prefix syntax to express the application of a binary operation. Infix operators usually make expressions easier to read; they also make statement of mathematical properties more convenient.

Often we use several infix operators in an expression. To ensure that the expression is not ambiguous (i.e., the operations are done in the desired order), we must either use parentheses to give the order explicitly (e.g., `((y * (z+2)) + x)`) or use syntactic conventions to give the order implicitly.

Typically the application order for adjacent operators of different kinds is determined by the relative *precedence* of the operators. For example, the multiplication (`*`) operation has a higher precedence (i.e., binding power) than addition (`+`), so, in the absence of parentheses, a multiplication will be done before an adjacent addition. That is, `x + y * z` is taken as equivalent to `(x + (y * z))`.

In addition, the application order for adjacent operators of the same binding power is determined by a *binding* (or *association*) order. For example, the addition (`+`) and subtraction `-` operations have the same precedence. By convention, they bind more strongly to the *left* in arithmetic expressions. That is, `x + y - z` is taken as equivalent to `((x + y) - z)`.

By convention, operators such as exponentiation (denoted by ˆ) and cons bind more strongly to the *right*. Some other operations (e.g., division and the relational comparison operators) have no default binding order–they are said to have *free* binding.

Accordingly, Haskell provides the statements `infix`, `infixl`, and `infixr` for declaring a symbol to be an infix operator with free, left, and right binding, respectively. The first argument of these statements give the precedence level as an integer in the range 0 to 9, with 9 being the strongest binding. Normal function application has a precedence of 10.

The operator precedence table for a few of the common operations from the Prelude is shown below. We introduce the `++` operator in the next subsection.

```
infixr 8 ^                     -- exponentiation
infixl 7 *                     -- multiplication
infix  7 /                     -- division
infixl 6 +, -                  -- addition, subtraction
infixr 5 :                     -- cons
infix  4 ==, /=, <, <=, >=, >  -- relational comparisons
infixr 3 &&                    -- Boolean AND
infixr 2 ||                    -- Boolean OR
```

4.5.1 Appending two lists: `++`

Suppose we want a function that takes two lists and returns their concatenation, that is, *appends* the second list after the first. This function is a binary operation on lists much like `+` is a binary operation on integers.

Further suppose we want to introduce the infix operator symbol `++` for the append function. Since we want to evaluate lists lazily from their heads, we choose right binding for both cons and `++`. Since append is, in a sense, an extension of cons (`:`), we give them the same precedence:

```
infixr 5 ++
```

Consider the definition of the append function. We must define the `++` operation in terms of application of already defined list operations and recursive applications of itself. The only applicable simpler operation is cons.

As with previous functions, we follow the type to the implementation–let the form of the data guide the form of the algorithm.

The cons operator takes an element as its left operand and a list as its right operand and returns a new list with the left operand as the head and the right operand as the tail.

Similarly, `++` must take a list as its left operand and a list as its right operand and return a new list with the left operand as the initial segment and the right operand as the final segment.

Given the definition of cons, it seems reasonable that an algorithm for `++` must consider the structure of its left operand. Thus we consider the cases for nil and non-nil left operands.

- If the left operand is nil, then the function can just return the right operand.
- If the left operand is a cons (that is, non-nil), then the result consists of the left operand's head followed by the append of the left operand's tail to the right operand.

In following the type to the implementation, we use the form of the left operand in a pattern match. We define `++` as follows:

```
infixr 5 ++

(++) :: [a] -> [a] -> [a]
[] ++ xs     = xs            -- nil left operand
(x:xs) ++ ys = x:(xs ++ ys) -- non-nil left operand
```

Above we use infix patterns on the left-hand sides of the defining equations.

For the recursive application of `++`, the length of the left operand decreases by one. Hence the left operand of a `++` application eventually becomes nil, allowing the evaluation to terminate.

Consider the evaluation of the expression `[1,2,3] ++ [3,2,1]`.

	`[1,2,3] ++ [3,2,1]`
$\Longrightarrow$	`1:([2,3] ++ [3,2,1])`
$\Longrightarrow$	`1:(2:([3] ++ [3,2,1]))`
$\Longrightarrow$	`1:(2:(3:([] ++ [3,2,1])))`
$\Longrightarrow$	`1:(2:(3:[3,2,1]))`
$=$	`[1,2,3,3,2,1]`

The number of steps needed to evaluate `xs ++ ys` is proportional to the length of `xs`, the left operand. That is, the time complexity is $O(n)$, where n is the length `xs`.

Moreover, `xs ++ ys` only needs to copy the list `xs`. The list `ys` is shared between the second operand and the result. If we did a similar function to append two (mutable) arrays, we would need to copy both input arrays to create the output array. Thus, in this case, a linked list is more efficient than arrays!

Consider the following questions:

- Is `++` tail recursive?
- What is the space complexity of `++`?

4.5.2 Properties of operations

The append operation has a number of useful algebraic properties, for example, associativity and an identity element.

Associativity of `++`: For any finite lists `xs`, `ys`, and `zs`, `xs ++ (ys ++ zs) == (xs ++ ys) ++ zs`.

Identity for `++`: For any finite list `xs`, `[] ++ xs = xs = xs ++ []`.

We will prove these and other properties in a later chapter.

Mathematically, the list data type and the binary operation `++` form a kind of abstract algebra called a *monoid.* Function `++` is closed (i.e., it takes two lists and gives a list back), is associative, and has an identity element. ‘ Similarly, we can state properties of combinations of functions. We can prove these using techniques we study in a later chapter. For example, consider the functions defined above in this chapter.

- For all finite lists `xs`, we have the following distribution properties:

  ```
  sum' (xs ++ ys)     = sum' xs + sum' ys
  product' (xs ++ ys) = product' xs * product' ys
  length' (xs ++ ys)  = length' xs + length' ys
  ```

- For all natural numbers `n` and finite lists `xs`,

  ```
  take n xs ++ drop n xs = xs
  ```

4.5.3 Element selection: `!!`

As another example of an infix operation, consider the list selection operator `!!`. The expression `xs!!n` selects element `n` of list `xs` where the head is in position 0. It is defined in the Prelude similar to the way `!!` is defined below:

```
infixl 9 !!

(!!) :: [a] -> Int -> a
xs     !! n | n < 0 = error "!! negative index"
[]     !! _         = error "!! index too large"
(x:_)  !! 0         = x
```

```
    (_:xs) !! n          = xs !! (n-1)
```

Consider the following questions concerning the element selection operator:

- What is the precondition for element selection?
- Does evaluation terminate?
- Is the operator tail recursive?
- Does the result share any data with the input list?
- What are its time and space complexities?

4.5.4 Reversing a list: `rev`

Consider the problem of reversing the order of the elements in a list.

Again we can use the structure of the data to guide the algorithm development. If the argument is nil, then the function returns nil. If the argument is non-nil, then the function can append the head element at the back of the reversed tail.

```
    rev :: [a] -> [a]
    rev []     = []             -- nil argument
    rev (x:xs) = rev xs ++ [x] -- non-nil argument
```

Given that evaluation of `++` terminates, we note that evaluation of `rev` also terminates because all recursive applications decrease the length of the argument by one.

How efficient is this function?

Consider the evaluation of the expression `rev "bat"`.

	`rev "bat"`
$\Longrightarrow$	`(rev "at") ++ "b"`
$\Longrightarrow$	`((rev "t") ++ "a") ++ "b"`
$\Longrightarrow$	`(((rev "") ++ "t") ++ "a") ++ "b"`
$\Longrightarrow$	`(("" ++ "t") ++ "a") ++ "b"`
$\Longrightarrow$	`("t" ++ "a") ++ "b"`
$\Longrightarrow$	`('t':("" ++ "a")) ++ "b"`
$\Longrightarrow$	`"ta" ++ "b"`
$\Longrightarrow$	`'t':("a" ++ "b")`
$\Longrightarrow$	`'t':('a':("" ++ "b"))`
$\Longrightarrow$	`'t':('a':"b")`
$=$	`"tab"`

The evaluation of `rev` takes $O(n^2)$ steps, where n is the length of the argument. There are $O(n)$ applications of `rev`; for each application of `rev` there are $O(n)$ applications of `++`.

The initial list and its reverse do not share data.

Function `rev` has a number of useful properties, for example the following.

Distribution: For any finite lists `xs` and `ys`, `rev (xs ++ ys) = rev ys ++ rev xs`.

Inverse: For any finite list `xs`, `rev (rev xs) = xs`.

Also, for any finite lists `xs` and `ys` and natural numbers `n`, we can state properties such as:

```
rev (xs ++ ys) = rev ys ++ rev xs
take n (rev xs)  = rev (drop (length xs - n) xs)
```

4.5.5 Tail recursive `reverse`

Most of the list function definitions examined so far are *backward recursive*. That is, for each case the recursive applications are embedded within another expression. Operationally, significant work is done after the recursive call returns.

Now let's look at the problem of reversing a list again to see whether we can devise a more efficient *tail recursive* solution.

As we have seen, the common technique for converting a backward linear recursive definition like `rev` into a tail recursive definition is to use an *accumulating parameter* to build up the desired result incrementally. A possible definition follows:

```
rev' [] ys     = ys
rev' (x:xs) ys = rev' xs (x:ys)
```

In this definition parameter `ys` is the accumulating parameter. The head of the first argument becomes the new head of the accumulating parameter for the tail recursive call. The tail of the first argument becomes the new first argument for the tail recursive call.

We know that `rev’` terminates because, for each recursive application, the length of the first argument decreases toward the base case of `[]`.

We note that `rev xs` is equivalent to `rev’ xs []`. (We can prove this using the techniques in a later chapter.)

To define a single-argument replacement for `rev`, we can embed the definition of `rev’` as an *auxiliary* function within the definition of a new function `reverse’`. (This is similar to function `reverse` in the Prelude.)

```
reverse' :: [a] -> [a]
reverse' xs = rev xs []
              where rev []     ys = ys
                    rev (x:xs) ys = rev xs (x:ys)
```

The `where` clause introduces the *local definition* `rev'` that is only known within the right-hand side of the defining equation for the function `reverse'`.

What is the time complexity of this function?

The evaluation of `reverse'` takes $O(n)$ steps, where n is the length of the argument. There is one application of `rev'` for each element; `rev'` requires a single cons operation in the accumulating parameter.

Where did the increase in efficiency come from?

Each application of `rev` applies `++`, a linear time (i.e., $O(n)$) function. In `rev'`, we replaced the applications of `++` by applications of cons, a constant time (i.e., $O(1)$) function.

In addition, a compiler or interpreter that does tail call optimization can translate this tail recursive call into a loop on the host machine.

4.6 More Useful List Functions

4.6.1 Another list-breaking function: `splitAt`

Above we defined list-breaking functions `take'` and `drop'`. It is sometimes useful to have a single function that breaks a list into two parts.

The function `splitAt` (shown below as `splitAt'`) takes an integer `n` and a list and returns a pair whose first component is the first `n` elements of the list and second component is the list remaining after the first `n` elements are removed.

```
splitAt' :: Int -> [a] -> ([a],[a])
splitAt' n xs =  (take' n xs, drop' n xs)
```

Can we write an alternative definition that makes only one pass over argument `xs`? (That is, it does not call `take'` and `drop'`.)

4.6.2 List-combining operations: `zip` and `unzip`

Another useful function in the Prelude is `zip` (shown below as `zip'`) which takes two lists and returns a list of pairs of the corresponding elements. That is, the function fastens the lists together like a zipper. It's definition is similar to `zip'` given below:

```
zip' :: [a] -> [b] -> [(a,b)]
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys -- zip.1
zip' _      _      = []                 -- zip.1
```

Function `zip` applies a *tuple-forming* operation to the corresponding elements of two lists. It stops the recursion when either list argument becomes nil. Putting the recursive case first enabled the two bases cases to be combined into one leg.

Example: `zip [1,2,3] "oxford"` $\Longrightarrow \cdots$ `[(1,’o’),(2,’x’),(3,’f’)]`

Similarly, function `unzip` in the Prelude takes a list of pairs and returns a pair of lists. It’s definition is similar to `unzip'` below.

```
unzip' :: [(a,b)] -> ([a],[b])
unzip' []         = ([],[])
unzip' ((x,y):ps) = (x:xs, y:ys)
                    where (xs,ys) = unzip' ps
```

The Prelude includes versions of `zip` and `unzip` that handle the tuple-formation for up to seven input lists: `zip3` $\cdots$ `zip7` and `unzip3` $\cdots$ `unzip7`.

4.7 Local Definitions

The `let` expression is useful whenever a nested set of definitions is required. It has the following syntax:

`let` *local_definitions* `in` *expression*

A `let` may be used anywhere that an expression my appear in a Haskell program.

For example, consider a function `f` that takes a list of integers and returns a list of their squares incremented by one:

```
f :: [Int] -> [Int]
f [] = []
f xs = let square a = a * a
           one      = 1
           (y:ys)   = xs
       in (square y + one) : f ys
```

- `square` represents a function of one variable.
- `one` represents a constant, that is, a function of zero variables.
- `(y:ys)` represents a pattern match binding against argument `xs` of `f`.
- Reference to `y` or `ys` when argument `xs` of `f` is nil results in an error.
- Local definitions `square`, `one`, `y`, and `ys` all come into scope simultaneously; their scope is the expression following the `in` keyword.
- Local definitions may access identifiers in outer scopes (e.g., `xs` in definition of `(y:ys)`) and have definitions nested within themselves.
- Local definitions may be recursive and call each other.

The `let` clause introduces symbols in a bottom-up manner: it introduces symbols before they are used.

The `where` clause is similar semantically, but it introduces symbols in a top-down manner: the symbols are used and then defined in a `where` that follows.

The `where` clause is more versatile than the `let`. It allows the scope of local definitions to span over several guarded equations while a `let`'s scope is restricted to the right-hand side of one equation.

For example, consider the definition:

```
g :: Int -> Int
g n | check3 == x = x
    | check3 == y = y
    | check3 == z = z * z
                where check3 = n `mod` 3
                      x = 0
                      y = 1
                      z = 2
```

- The scope of this `where` clause is over *all three guards* and their respective right-hand sides. (Note that the `where` begins in the same column as the `=` rather than to the right as in `rev'`.)
- Note the use of the modulo function `mod` as an infix operator. The backquotes (`‘`) around a function name denotes the infix use of the function.

In addition to making definitions easier to understand, local definitions can increase execution efficiency in some cases. A local definition may introduce a component into the expression graph that is shared among multiple branches. Haskell uses graph reduction, so any shared component is evaluated once and then replaced by its value for subsequent accesses.

The local variable `check3` introduces a component shared among all three legs. It is evaluated once for each call of `g`.

4.8 Insertion Sort

Consider a function to sort the elements of a list into ascending order.

A list is *ascending* if every element is `<=` all of its successors in the list. Successor means an element that occurs later in the list (i.e., away from the head). A list is *increasing* if every element is `<` its successors. Similarly, a list is *descending* or *decreasing* if every element is `>=` or `>`, respectively, its successors.

A simple algorithm to do this is *insertion sort*. To sort a non-empty list with head `x` and tail `xs`, sort the tail `xs` and then insert the element `x` at the right position in the result. To sort an empty list, just return it.

If we restrict the function to integer lists, we get the following Haskell functions:

```
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = insert x (isort xs)
```

```
insert :: Int -> [Int] -> [Int]
insert x []     = [x]
insert x xs@(y:ys)
    | x <= y    = (x:xs)
    | otherwise = y : (insert x ys)
```

Insertion sort has a (worst and average case) time complexity of O(n^2) where n is the length of the input list. (Function `isort` requires n consecutive recursive calls; each call uses function `insert` which itself requires on the order of n recursive calls.)

Now suppose we want to generalize the sorting function and make it polymorphic. We cannot just add a type parameter `a` and substitute it for `Int` everywhere. Not all Haskell types can be compared on a *total ordering* (`<`, `<=`, `>`, and `>=` as well).

We need to constrain the polymorphism to types in class `Ord`, as follows:

```
isort' :: Ord a => [a] -> [a]
isort' []     = []
isort' (x:xs) = insert' x (isort' xs)

insert' :: Ord a => a -> [a] -> [a]
insert' x []    = [x]
insert' x xs@(y:ys)
    | x <= y    = (x:xs)
    | otherwise = y : (insert' x ys)
```

We could define `insert'` inside `isort'` and avoid the separate type parameterization. But `insert` is separately useful, so it is reasonable to leave it external.

Consider the following questions:

- How do we know `insert'` terminates?
- What are the time and space complexities of `insert'`?
- How do we know `isort'` terminates?
- What are the time and space complexities of `isort'`?

4.9 Exercises

1. Write a Haskell function to compute the maximum value in a nonempty list of integers. Generalize the function by making it polymorphic, accepting a value from any ordered type.

2. Write a Haskell function `adjpairs` that takes a list and returns the list of all pairs of adjacent elements. For example, `adjpairs [2,1,11,4]` returns `[(2,1), (1,11), (11,4)]`.

3. Write a Haskell function `mean` that takes a list of integers and returns the mean (i.e., average) value for the list.

4. Write tail recursive versions of the Haskell functions `sum'` and `product'`.

5. Answer the following questions for the `take'` function defined in this chapter:

   - What is returned when the list argument is nil?
   - Does evaluation of the function terminate?
   - Does the result share data with the input?
   - Is the function tail recursive?
   - What are its time and space complexities?

6. Answer the following question for the `++` operator developed in this chapter.

   - Is `++` tail recursive?
   - What is the space complexity of `++`?

7. Answer the following questions concerning the element selection operator defined in this chapter.

   - What is the precondition for element selection?
   - Does evaluation terminate?
   - Is the operator tail recursive?
   - Does the result share any data with the input list?
   - What are its time and space complexities?

8. Write a version of function `splitAt'` that makes only one pass over the input list (that is, does not call `take'` and `drop'`).

9. Answer the following questions for the `isort'` and `insert'` functions.

   - How do we know `insert'` terminates?
   - What are the time and space complexities of `insert'`?
   - How do we know `isort'` terminates?
   - What are the time and space complexities of `isort'`?

10. Hailstone functions.

    a. (This part is repeated from a previous chapter.) Develop a function `hailstone` to implement the following function:

$$\begin{array}{lcll}
\hline
hailstone(n) & = & 1, & \text{if } n = 1 \\
hailstone(n) & = & hailstone(n/2), & \text{if } n > 1, \text{ even } n \\
hailstone(n) & = & hailstone(3 * n + 1), & \text{if } n > 1, \text{ odd } n \\
\hline
\end{array}$$

    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?

b. Write a Haskell function that computes the results of the `hailstone` function for each element of a list of positive integers. The value returned by the `hailstone` function for each element of the list should be displayed.

c. Modify the `hailstone` function to return the function's "path." That is, each application of this path function should return a list of integers instead of a single integer. The list returned should consist of the arguments of the successive calls to the `hailstone` function necessary to compute the result. For example, the `hailstone 3` example above should return `[3,10,5,16,8,4,2,1]`.

11. Number base conversion.

    a. Write a Haskell function `natToBin` that takes a natural number and returns its binary representation as a list of `0`'s and `1`'s with the most significant digit at the head. For example, `natToBin 23` returns `[1,0,1,1,1]`. (Note: Prelude function `rem` returns the remainder from dividing its first argument by its second. Enclosing the function name in backquotes as in `‘rem‘` allows a two-argument function to be applied in an infix form.)

    b. Generalize `natToBin` to function `natToBase` that takes a base `b` (`b \geq 2`) and a natural number and returns the base `b` representation of the natural number as a list of integer digits with the most significant digit at the head. For example, `natToBase 5 42` returns `[1,3,2]`.

    c. Write Haskell function `baseToNat`, the inverse of the `natToBase` function. For any base `b` (`b \geq 2`) and natural number `n`:

    ```
    baseToNat b (natToBase b n) = n
    ```

12. Write a Haskell function `merge` that takes two increasing lists of integers and merges them into a single increasing list (without any duplicate values). A list is *increasing* if every element is less than (`<`) its successors. Successor means an element that occurs later in the list, i.e., away from the head. Generalize the function by making it polymorphic.

13. Design a module of set operations. Choose a Haskell representation for sets. Implement functions to make sets from lists and vice versa, to insert

and delete elements from sets, to do set union, intersection, and difference, to test for equality and subset relationships, to determine cardinality, and so forth.

14. Bag operation package.

    A *bag* (or multiset) is a collection of elements; each element may occur one or more times in the bag. Choose an efficient representation for bags. Each bag should probably have a unique representation.

    Define a package of bag operations, including the following functions. For the functions that return bags, make sure that a valid representation of the bag is returned.

    - `listToBag` takes a list of elements and returns a bag containing exactly those elements. The number of occurrences of an element in the list and in the resulting bag is the same.
    - `bagToList` takes a bag and returns a list containing exactly the elements occurring in the bag. The number of occurrences of an element in the bag and in the resulting list is the same.
    - `bagToSet` takes a bag and returns a list containing exactly the *set* of elements contained in the bag. Each element occurring one or more times in the bag will occur exactly once in the list returned.
    - `bagEmpty` takes a bag and returns `True` if the bag is empty and `False` otherwise.
    - `bagCard` takes a bag and returns its cardinality (i.e., the total number of occurrences of all elements).
    - `bagElem` takes an element and a bag and returns `True` if the element occurs in the bag and `False` otherwise.
    - `bagOccur` takes an element and a bag and returns the number of times the element occurs in the bag.
    - `bagEqual` takes two bags and returns `True` if the two bags are equal (i.e., the same elements and same number of occurrences of each) and `False` otherwise.
    - `bagSubbag` takes two bags and returns `True` if the first is a subbag of the second and `False` otherwise. X is a subbag of Y if every element of X occurs in Y at least as many times as it does in X.
    - `bagUnion` takes two bags and returns their bag union. The union of bags X and Y contains all elements that occur in either X or Y; the number of occurrences of an element in the union is the number in X or in Y, whichever is greater.
    - `bagIntersect` takes two bags and returns their bag intersection. The intersection of bags X and Y contains all elements that occur in both

X and Y; the number of occurrences of an element in the intersection is the number in X or in Y, whichever is lesser.

- `bagSum` takes two bags and returns their bag sum. The sum of bags X and Y contains all elements that occur in X or Y; the number of occurrences of an element is the sum of the number of occurrences in X and Y.

- `bagDiff` takes two bags and returns the bag difference, first argument minus the second. The difference of bags X and Y contains all elements of X that occur in Y fewer times; the number of occurrences of an element in the difference is the number of occurrences in X minus the number in Y.

- `bagInsert` takes an element and a bag and returns the bag with the element inserted. Bag insertion either adds a single occurrence of a new element to the bag or increases the number of occurrences of an existing element by one.

- `bagDelete` takes an element and a bag and returns the bag with the element deleted. Bag deletion either removes a single occurrence of the element from the bag or decreases the number of occurrences of the element by one.

15. Unbounded precision arithmetic package for natural numbers (i.e., non-negative integers). Do not use the builtin `Integer` type.

    a. Define a type synonym `BigNat` to represent these unbounded precision natural numbers as lists of `Int`. Let each element of the list denote a decimal digit of the "big natural" number represented, with the *least* significant digit at the head of the list and the remaining digits given in order of *increasing* significance. For example, the integer value 22345678901 is represented as `[1,0,9,8,7,6,5,4,3,2,2]`. Use the following "canonical" representation: the value `0` is represented by the list `[0]` and positive numbers by a list without "leading" `0` digits (i.e., `126` is `[6,2,1]` not `[6,2,1,0,0]`). You may use the nil list `[ ]` to denote an error value.

       Define a package of basic arithmetic operations, including the following functions. Make sure that `BigNat` values returned by these functions are in canonical form.

       - `intToBig` takes a nonnegative `Int` and returns the `BigNat` with the same value.

       - `strToBig` takes a `String` containing the value of the number in the "usual" format (i.e., decimal digits, left to right in order of *decreasing* significance with zero or more leading spaces, but with no spaces or punctuation embedded within the number) and returns the `BigNat` with the same value.

    - `bigToStr` takes a `BigNat` and returns a `String` containing the value of the number in the "usual" format (i.e., left to right in order of *decreasing* significance with no spaces or punctuation).
    - `bigComp` takes two `BigNat`s and returns the `Int` value `-1` if the value of the first is less than the value of the second, the value `0` if they are equal, and the value `1` if the first is greater than the second.
    - `bigAdd` takes two `BigNat`s and returns their sum as a `BigNat`.
    - `bigSub` takes two `BigNats` and returns their difference as a `BigNat`, first argument minus the second.
    - `bigMult` takes two `BigNats` and returns their product as a `BigNat`.

  b. Use the package to generate a table of factorials for the naturals 0 through 25. Print the values from the table in two *right-justified* columns, with the number on the left and its factorial on the right. (Allow about 30 columns for `25!`.)

  c. Use the package to generate a table of Fibonacci numbers for the naturals 0 through 50.

  d. Generalize the package to handle signed integers. Add the following new function:

    - `bigNeg` returns the negation of its `BigNat` argument.

  e. Add the following functions to the package:

    - `bigDiv` takes two `BigNat`s and returns, as a `BigNat`, the quotient from dividing the first argument by the second.
    - `bigRem` takes two `BigNat`s and returns, as a `BigNat`, the remainder from dividing the first argument by the second.

16. Define the following set of text-justification functions. You may want to use Prelude functions like `take`, `drop`, and `length`.

    - `spaces' n` returns a string of length `n` containing only space characters (i.e., the character `' '`).
    - `left' n xs` returns a string of length `n` in which the string `xs` begins at the head (i.e., left end).

      Examples: left' 3 "ab" yields`"ab "; left' 3 "abcd" yields "abc".
    - `right' n xs` returns a string of length `n` in which the string `xs` ends at the tail (i.e., right end).

      Examples: `right' 3 bc` yields `bc`; `right' 3 abcd` yields `bcd`.

- `center' n xs` returns a string of length `n` in which the string `xs` is approximately centered.

  Example: `center' 4 "bc"` yields `" bc "`.

17. Consider simple mathematical expressions consisting of integer constants, variable names, parentheses, and the binary operators `+`, `-`, `*`, and `/`. For the purposes of this exercise, an *expression* is a string that satisfies the following (extended) BNF grammar and lexical conventions:

    - The characters in an input string are examined left to right to form "lexical tokens". The tokens of the expression "language" consist of *addOp*s, *mulOp*s,*identifier*s, *number*s, and left and right parentheses.
    - An expression may contain space characters at any position except within a lexical token.
    - An *addOp* token is either a "`+`" or a "`-`"; a *mulOp* token is either a "`*`" or a "`/`".
    - An *identifier* `q` is a string of one or more contiguous characters such that the leftmost character is a letter and the remaining characters are either letters, digits, or underscore characters.

      Examples: "`Hi1`", "`lo23_1`", "`this_is_2_long`"
    - A *number* is a string of one or more contiguous characters such that all (including the leftmost) are digits.

      Examples: "`1`", "`23456711`"
    - All *identifier* and *number* tokens extend as far to the right as possible. For example, consider the string "`A123  12B3+2 )`". (Note the space and right parenthesis characters). This string consists of the six tokens "`A123`", "`12`", "`B3`", "`+`", "`2`", and "`)`".

    Define a Haskell function `valid` that takes a `String` and returns `True` if the string is an *expression* as described above and returns `False` otherwise.

    Hints:

    - If you need to return more than one value from a function, you can do so by returning a tuple of those values. This tuple can be decomposed by Prelude functions such as `fst` and `snd`.
    - Use of the `where` or `let` features can simplify many functions. You may find Prelude functions such as `span`, `isSpace`, `isDigit`, `isAlpha`, and `isAlphanum` useful.
    - You may want to consider organizing your program as a simple *recursive descent* recognizer for the expression language.

18. Extend the mathematical expression recognizer of the previous exercise to *evaluate* integer expressions with the given syntax. The four binary operations have their usual meanings.

    Define a function `eval e st` that evaluates expression `e` using symbol table `st`. If the expression `e` is syntactically valid, `eval` returns a pair `(True,val)` where `val` is the value of `e`. If `e` is not valid, `eval` returns `(False,0)`.

    The symbol table consists of a list of pairs, in which the first component of a pair is the variable name (a string) and the second is the variable's value (an integer).

    Example: `eval "(2+x) * y" [("y",3),("a",10),("x",8)]` yields `(True,30)`.

4.10 References

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

[**Chiusano-Bjarnason 2015**]] Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015.

[**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.

4.11 Terms and Concepts

Polymorphism (ad hoc, overloading, subtyping, parametric), lists (polymorphic, immutable, persistent, data sharing, empty, nonempty), list and string operations (cons, head, tail, pattern matching), syntactic sugar, type synonym, type variable, follow the types to implementations, let the form of the data guide the form of the algorithm, infix operation, properties of operators (associative, identity, zero, inverse, distribution), precedence (left, right, free binding), context predicates (type classes `Eq`, `Ord`), insertion sort