Exploring Languages with Interpreters and Functional Programming Chapter 14

H. Conrad Cunningham

17 October 2018

Contents

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

Browser Advisory: The HTML version of this textbook requires use of a browser that supports the display of MathML. A good choice as of October 2018 is a recent version of Firefox from Mozilla.

14 Infix Operators and List Examples

14.1 Chapter Introduction

This chapter introduces Haskell infix operations and continues to develop techniques for first-order polymorphic functions to process lists.

The goals of the chapter are to:

- introduce Haskell syntax and semantics for infix operations
- examine correct Haskell functional programs consisting of first-order polymorphic functions that solve problems by processing lists and strings
- explore methods for developing Haskell list-processing programs that terminate and are efficient and elegant.

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

14.2 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 `((x + y) - z)`.

By convention, operators such as exponentiation (denoted by `^`) 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
```

14.2.1 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]
⟹    1:([2,3] ++ [3,2,1])
⟹    1:(2:([3] ++ [3,2,1]))
⟹    1:(2:(3:([] ++ [3,2,1])))
⟹    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:

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

14.2.2 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 Chapter 25.

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
```

14.2.3 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?

14.2.4 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"
⟹   (rev "at") ++ "b"
⟹   ((rev "t") ++ "a") ++ "b"{.haskell}
⟹   (((rev "") ++ "t") ++ "a") ++ "b"
⟹   (("" ++ "t") ++ "a") ++ "b"
⟹   ("t" ++ "a") ++ "b"
⟹   ('t':("" ++ "a")) ++ "b"
⟹   "ta" ++ "b"
⟹   't':("a" ++ "b")
⟹   't':('a':("" ++ "b"))
⟹   '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)
```

14.2.5 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.

14.3 More Useful List Functions

14.3.1 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'`.)

14.3.2 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 triples. Librart `Data.List` includes functions for up to seven input lists: `zip4` $\cdots$ `zip7` and `unzip4` $\cdots$ `unzip7`.

14.4 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'`?

14.5 What Next?

This and the preceding chapter explored use of first-order polymorphic functions to process lists in Haskell.

The next three chapters examine higher-order function concepts in Haskell.

14.6 Exercises

1. Answer the following questions for the `++` operation defined in this chapter:
   - Is `++` tail recursive?
   - What is the space complexity?

1. 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?

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

3. 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'`?

4. Hailstone functions.
   a. (This part is repeated from a previous chapter.) 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

$$hailstone(n) \quad = \quad hailstone(3 * n + 1), \quad \text{if } n > 1, \text{ odd } n$$

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?

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]`.

5. 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
```

6. 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.

7. 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.

8. Bag module.

   Mathematically, a *bag* (or *multiset*) is a function from some arbitrary set of elements (the domain) to the set of nonnegative integers (the range). We interpret the nonnegative integer as the number of occurrences of the element in the bag. Zero means the element does not occur.

   From another perspective, a bag is an unordered collection of elements. Each element may occur one or more times in the bag. (It is like a set except values can occur multiple times.)

   For example, `{| "time', "time", "and", "time", "again" |}` is a bag containing 5 strings. There are 3 occurrences of string `"time"` and 1 occurrence each of strings `"and"` and `"again"`.

   `{| 11, 2, 3, 7, 5 |}` is a bag of prime numbers. It is also a set because each element occurs exactly once.

   We can represent a bag in many ways in Haskell. Using lists, we could represent a bag with a simple (unordered) list of elements, an ordered list of elements, an unordered or an ordered list of tuples which pair an element with the (nonzero) number of times it occurs, etc. A bag could also be represented with other data structures such as a `Map` from library `Data.Map`.

   Choose some representation for polymorphic bags. You may assume that the elements in the domain are totally ordered (i.e. are from a type that is an instance of class `Ord`), but otherwise the elements can be of any type.

   For example, if you use a list representation, you might define the type synonym:

   ```
   type Bag a = [a]
   ```

   Develop a data abstraction (information-hiding) module that encapsulates the representation of the data structure used to store the elements inside the module.

The module should include the following public functions. This interface should be the same even if you change the representation of the data internally.

a. `newBag` returns a new bag with no elements (i.e. empty).

b. `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.

c. `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.

   Note: It is not required that:

   ```
   bagToList (listToBag xs) == xs
   ```

   But it is required that both sides have the same numbers of the same elements.

d. `isEmpty` takes a bag and returns `True` if the bag has no elements and returns `False` otherwise.

e. `isElem` takes an element and a bag and returns `True` if the element occurs in the bag and returns `False` otherwise.

f. `size` takes a bag and returns its cardinality (i.e. the total number of occurrences of all elements).

g. `occursBag` takes an element and a bag and returns the number of occurrences of the element in the bag.

h. `insertElem` 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.

i. `deleteElem` takes an element and a bag and returns the bag with the element deleted. Bag deletion removes a single occurrence of an element from the bag, decreases the number of occurrences of an existing element by one, or does not change the bag if the element does not occur.

j. `eqBag` 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 returns `False` otherwise.

   Note: If `bagToList xs == bagToList ys`, then `eqBag xs ys`. However, if `eqBag xs ys`, it is not required that `bagToList xs == bagToList ys`.

k. `unionBag` 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.

l. `intersectBag` 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.

m. `sumBag` 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.

n. `diffBag` 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.

o. `subBag` 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.

p. `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.

9. Develop a bag module as described in the previous exercise, but use a different internal representation than you used in the previous exercise. The new module should have the same public interface as the previous module.

10. Unbounded precision arithmetic module for natural numbers (i.e. nonnegative 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 Haskell module with 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 `BigNat` s and returns their difference as a `BigNat`, first argument minus the second.
- `bigMult` takes two `BigNat`s 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.

11. 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 "`.

12. 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.

13. 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)`.

14.7 Acknowledgements

In Summer 2016, I adapted and revised much of this work from the following sources:

- chapter 5 of my *Notes on Functional Programming with Haskell* [Cunningham 2014] which is influenced by [Bird 1988] (later editions are [Bird 1998] and [Bird 2015])
- my notes on *Functional Data Structures (Scala)* [Cunningham 2016] which are based, in part, on chapter 3 of the book *Functional Programming in Scala* [Chiusano 2015].

In 2017, I continued to develop this work as Chapter 4, List Programming, of my 2017 Haskell-based programming languages textbook.

In Summer 2018, I divided the previous List Programming chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Previous sections 4.1-4.4 became the basis for new Chapter 13, List Programming, and previous sections 4.5-4.8 became the basis for Chapter 14, Infix Operators and List Programming Examples (this chapter).

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

14.8 References

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

[Bird 1998]: Richard Bird. *Introduction to Functional Programming using Haskell*, Second Edition, Prentice Hall, 1998.

[Bird 2015]: Richard Bird. *Thinking Functionally with Haskell*, Second Edition, Cambridge University Press, 2015.

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

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

[Cunningham 2016]: H. Conrad Cunningham, *Functional Data Structures (Scala)*, 2016. (Lecture notes based, in part, on chapter 3 [Chiusano 2015].)

14.9 Terms and Concepts

Binary operation, infix operation, properties of operators (associative, identity, zero, inverse, distribution), precedence (left, right, free binding).