A course on concepts of programming languages:

notes and programming problems

Andrei Lapets

January 21, 2009

This is a collection of notes about fundamental concepts in programming language design, with particular emphasis on functional programming techniques and paradigms. The notes utilize the functional language Haskell in examples, and contain problems to be solved by writing Haskell programs. These notes and problems were initially written as section notes and homework assignments for the fall of 2007 and fall of 2008 incarnations of an advanced undergraduate course entitled "Concepts of Programming Languages" (BU, CAS CS 320), taught by Professor Assaf Kfoury.

Full solutions to all programming problems are available. They can be obtained by individual request.

Contents

Chapter 1

Recursive Functions and Infinite Lists

1.1 Tetranacci Numbers and Memoization

In this assignment, you will write several functions for computing a generalization of the Fibonacci numbers. All your solutions should be defined within a module named `Tetra`, and you should submit the file `Tetra.hs` (or `Tetra.lhs`). **File names are case sensitive.** Verbal responses to non-programming questions should be in the form of comments in your code.

Note: The solution to each part of every problem is meant to be extremely short (one to three lines). You may (and should) use functions from earlier problems to solve parts of later problems.

Problem 1.1.1. (15 pts)

The formula for the nth Tetranacci number T_n is defined as follows:

$$
\begin{aligned}
T_0 &= 0 \\
T_1 &= 1 \\
T_2 &= 1 \\
T_3 &= 2 \\
T_n &= T_{n-1} + T_{n-2} + T_{n-3} + T_{n-4}.
\end{aligned}
$$

Implement a recursive Haskell function `tetra1` that accepts an integer n (you may assume that $n \geq 0$), and computes the nth Tetranacci number (don't worry about efficiency, only about making the definition as simple as possible). In your code, specify a type for the function which ensures that it accepts and returns only arguments of type `Int`. What is the approximate number of times this function will be called on an input of size n?

Problem 1.1.2. (25 pts)

You will now implement a more efficient version of the function for computing Tetranacci numbers.

(a) Implement a Haskell function `sumtop4` that takes a list of integers and adds the sum of the top four elements to the head of the list (e.g. `1:1:1:1:nil` should become `4:1:1:1:1:nil`).

(b) Implement a recursive Haskell function `ascending` that accepts an integer n as input (again, assume $n \geq 0$), and returns a list of integers from 0 to n in ascending order.

(c) Implement a recursive Haskell function `tetra2` that computes the nth Tetranacci number in linear time. This function may use a linear amount of space to compute its result, but the number of recursive calls it makes must be linear in n.

Problem 1.1.3. (30 pts)

The Fibonacci k-step numbers are a generalization of the Fibonacci and Tetranacci sequences, where $F_i = 0$ for $i \leq 0$, $F_1 = 1$, $F_2 = 1$, and F_j for $j \geq 3$ is the sum of the k previous numbers in the sequence. You will implement a Haskell function for computing these numbers efficiently.

(a) Implement a Haskell function `sumtopk` that accepts an integer k along with a list of integers, and returns the sum of the first k elements of the list. If the list has length less than k, simply sum all the elements in the list (assume that an empty list evaluates to a sum of 0).

(b) Implement a function `fibkstep` that accepts two integers n and k, and computes in time $O(nk)$ the nth Fibonacci k-step number.

Problem 1.1.4. (30 pts)

For this problem, recall that because Haskell is a lazy language, an expression will not be evaluated before it is necessary to do so.

(a) Implement a Haskell function `listofints` that returns an infinite list of integers, in ascending order. (Hint: Will we need a base case in the function definition?)

(b) Implement a Haskell function `restAftern` that takes an integer n, calls and obtains the result from the function you wrote in part (a), throws away the first n elements of the result, and returns the remainder of the list. Does this function terminate for any n? What would happen if we tried to return the length of the resulting list? Why?

(c) With the help of the function from part (a), implement a function `inffib` that returns an infinite list of Fibonacci numbers (don't worry about efficiency).

1.2 Program Proof that the Set of Primes is Infinite

In this assignment, you will write a program which proves that there is an infinite number of primes. All your solutions should be defined within a module named `Primes`, and you should submit the file `Primes.hs` (or `Primes.lhs`). **File names are case sensitive.** Verbal responses to non-programming questions should be in the form of comments in your code.

Note: The solution to each part of every problem is meant to be extremely short (one to three lines). You may (and should) use functions from earlier problems to solve parts of later problems.

Problem 1.2.1. (10 pts)

Define a recursive function `prod` which takes a list of integers and computes the product of all the integers in that list. The product of an empty list of integers is defined to be 1. Only explicitly recursive solutions will receive credit. In your code, specify a type for the function which ensures that it accepts and returns only arguments of type `Int`.

Problem 1.2.2. (30 pts)

(a) Define a recursive function `ascending` which accepts two integers m and n as input, and returns the list of integers from m to n (including m and n) in ascending order. If $m > n$, your function should return the empty list.

(b) Recall that because Haskell is a lazy language, an expression will not be evaluated before it's necessary to do so. Define a recursive function `ascendingFrom` which takes a single integer argument m, and returns an infinite list of integers in ascending order, starting with m. (Hint: Will you need a base case in the function definition?)

(c) Define a function `naturals` which takes no arguments, and returns an infinite list of integers in ascending order, starting with 1.

Problem 1.2.3. (20 pts)

(a) For a positive integer n, a positive integer k is a factor of n if $n \bmod k = 0$ (in other words, if n divided by k leaves no remainder). Define a function `factors` which takes a single positive integer argument n, and returns a list containing all of its positive factors (including n itself). (Hint: You need a recursive helper function.)

(b) A number is *prime* if its only factors are one and itself. Using `factors`, define a function `isPrime` which takes a single positive integer argument n and returns `True` only if n is prime (returning `False` otherwise). Only solutions which make use of `factors` will receive credit.

Problem 1.2.4. (40 pts)

(a) Given your definitions from the problems above, we can define the following Haskell function, which returns a list of all primes:

```
primes = filter isPrime naturals
```

However, this definition does not actually prove that there is an infinite number of primes. Suppose there is only a *finite* number of primes. What can you then say about how the function `primes` will behave? If we do not observe this behavior, can we still be sure that we will *never* observe it?

(b) Using `factors`, define a function `primeFactors` which takes a single integer n and returns a list of its *prime* factors (including just itself, if it happens to be prime). (Hint: You only need to make a slight modification to the above definition of `primes` to accomplish this.)

(c) Euclid showed that given a list of primes $p_1, \ldots, p_n$, the value $(p_1 \cdot \ldots \cdot p_n) + 1$ (the product of this list of primes, plus one) is either itself a prime, or can be factored by a new prime which is *not* in this list. Define a function `anotherPrime` which takes a list of prime integers, and returns a single *new* prime integer which is not in this list. (Hint: use `prod` and `primeFactors`.)

(d) Define a recursive function `getprimes` which takes a single positive integer k as an argument, and returns a list with exactly k distinct prime integers in it. If $k = 0$, it should return an empty list. You should *not* need more than one base case if you defined `prod` according to the specification in **Problem #1.2.1**.

You have now defined a function which can return a list of distinct prime numbers for any arbitrarily large value. This means that if someone claims that there are only k primes for any finite k, your function can provide a list of $k+1$ primes which contradicts this claim, so there must be infinitely many distinct prime numbers.

Chapter 2

Standard List Functions

2.1 Implementing Simple Sets Using Lists

In this assignment, you will implement a module `Set` for representing sets of any type on which `(==)` is defined. You should submit a single file, `Set.hs` or `Set.lhs`, that contains a module definition beginning with:

```
module Set
  where

type Set a = [a]

-- function definitions start here.
```

All functions should be defined inside this module.

Problem 2.1.1. (15 pts)

As the above code suggests, you will implement sets using Haskell lists. In this problem, you may use built-in list functions wherever you see fit.

(a) Define a value `emp ::  Set a` which will represent an empty set.

(b) Define a function `e ::  Eq a => a -> Set a -> Bool` which takes an element `x` and a set which contains elements of that type, and returns `True` if and only if an element equal (that is, `(==)`) to `x` is in that set.

For later problems, you may find it convenient to add the following definition after you define `e`:

```
c :: Eq a => Set a -> a -> Bool c s x = x ‘e‘ s
```

(c) Define a function `add ::  Eq a => a -> Set a -> Set a` which adds an element `x` to a set *only if* an equivalent element is not already in the set. If there is already an equivalent element in the set, the original set should be returned. This ensures that sets will not contain duplicate elements. Note: This function must *not* be recursive.

(d) Define a function `size ::  Set a -> Int` which returns the size of a set.

Problem 2.1.2. (40 pts)

The functions implemented in this problem must *not* be recursive, recursive solutions will get no credit.

The built-in function `map ::  (a -> b) -> [a] -> [b]` simply applies a function to all the elements in a list, returning the resulting list, e.g.

$$\texttt{map not [True,True,False]} \implies \texttt{[False,False,True]}.$$

(a) Using `map`, implement a function `allIn ::  Eq a => Set a -> Set a -> [Bool]` which takes two sets, and replaces all the elements of the first set with boolean values that represent whether the element which was in that position was also in the second set, e.g.

$$\texttt{allIn [1, 2] [2, 3, 4]} \implies \texttt{[False,True]}.$$

The built-in function `foldr ::  (a -> b -> b) -> b -> [a] -> b` [HCFP, p 162] can be used in place of recursion to perform many common operations on lists. For example, it is possible to add all the elements in a list:

$$\texttt{foldr (+) 0 [1,2,3,4]} \implies \texttt{1 + (2 + (3 + (4 + 0)))} \implies \texttt{10}.$$

The first argument to `foldr` is a function `a -> b -> b` which operates on the *current* element of the list as well as the "running result." The second argument is the initial value to pass to this function, and the third argument is the list from which we'll draw each subsequent element (starting from the end and working towards the head of the list). Another example:

$$\texttt{foldr (:) [] [1,2,3,4]} \implies \texttt{1:(2:(3:(4:[])))} \implies \texttt{[1,2,3,4]}.$$

(b) Using `allIn`, `foldr`, and `(&&):  Bool -> Bool -> Bool`, implement a function `subset ::  Eq a => Set a -> Set a -> Bool` which returns true if and only if every element in the first set is also in the second set.

(c) Implement a function `union ::  Eq a => Set a -> Set a -> Set a` which returns a set containing all the elements in the two arguments. Duplicate elements are not allowed. (Hint: Don't forget `add` from the previous problem.)

The built-in function `filter ::  (a -> Bool) -> [a] -> [a]` takes a function and applies it to each element of a list. Every element for which the function returns `False` is thrown out of the list, e.g.

$$\text{filter ((>) 3) [1,2,3,4,5]} \implies \text{[1,2]}.$$

(d) Using `filter`, implement a function `isect ::  Eq a => Set a -> Set a -> Set a` which returns a set containing only elements which are found in *both* sets.

(e) Use `foldr` to implement a function `unionList ::  Eq a => [Set a] -> Set a` which returns the union of a list of sets. (Hint: You might want to use `emp`.)

(f) Use `foldr` to implement a function `isectList ::  Eq a => [Set a] -> Set a` which returns the intersection of a list of sets. (Hint: You might want to use `unionList` for the base case.)

Problem 2.1.3. (45 pts)

The power set of a set contains every subset of that set. For example, the power set of $\{1, 2, 3\}$ is $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}\}$. In this problem, you will add a function `powerset ::  Set a -> Set (Set a)` to the Set module which returns the power set of a set, e.g.

$$\text{powerset [1,2,3]} \implies \text{[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]}.$$

You will need to use the following function, which takes an integer and returns its binary representation in the form of a list of booleans:

```
decToBin :: Int -> [Bool] decToBin n
 | n == 0   = []
 | n > 0    = (mod2 n) : decToBin (div n 2)
  where
     mod2 n = if mod n 2 == 1 then True else False
```

(a) Implement a function `listOfBinInts ::  Int -> [[Bool]]` which takes an argument n and returns a list of integers from 0 to n in *in their [Bool] representation*. (Hint: This can be done in one line using list comprehensions, `decToBin`, and `map`).

The built-in function `zip ::  [a] -> [b] -> [(a,b)]` takes two lists (of not necessarily the same length) and turns it into a list of pairs, e.g.

$$\text{zip [1, 2, 3] ['a', 'b']} \implies \text{[(1,'a'),(2,'b')]}.$$

The built-in function `unzip ::  [(a,b)] -> ([a],[b])` does the opposite, e.g.

$$\text{unzip [(1,True), (2,False)]} \implies \text{([1,2],[True,False])}.$$

(b) Using `listOfBinInts`, implement a function `zipWithEach ::  Set a -> [[(a,Bool)]]` which builds a list with each and every "integer" (in `[Bool]` representation) from `0` to `(2 ^ (size s) - 1)`, and zips each of those "integers" with the set `s`. (Hint: Use `map` to `zip` each "integer" with the set `s`.)

Remark: Given any set of size n, each of the n items in the set can either be kept (`True`) or thrown out (`False`)). Thus, each of the 2^n integers in binary notation represents a possible combination of elements we could take out of our set. We zip each of these binary integers with our set, thus making 2^n copies of our set, each with a unique labelling of which elements to throw out, and which elements to keep. Notice that the result from `zipWithEach` almost gives us the power set:

```
zipWithEach [1,2,3]  ⟹  [[],
                         [(1,True)],
                         [(1,False),(2,True)],
                         [(1,True),(2,True)],
                         [(1,False),(2,False),(3,True)],
                         [(1,True),(2,False),(3,True)],
                         [(1,False),(2,True),(3,True)],
                         [(1,True),(2,True),(3,True)]].
```

It remains to throw out any pairs with `False` in the second field of the tuple, and to then convert the remaining tuples of the form `(?, True)` into single values.

(c) Implement a function `filterBySnd ::  [(a, Bool)] -> [(a, Bool)]` which throws out all tuples in a list where the second component is `False`.

(d) Implement a function `unzipFst ::  [(a, Bool)] -> [a]` which converts all the tuples in a list into values, throwing out the second `Bool` component.

(e) Use all of the functions you've defined, along with `map`, to implement `powerset :: Set a -> Set (Set a)`.

2.2   Representing Streams Using Infinite Lists

Potentially infinite lists are usually called "streams," or "sequences" when also referring to input/output channels. We will call them "sequences" throughout this assignment. More precisely, a sequence is a countable list of elements all of the same type; "countable" means the elements can be put in a $1 - 1$ correspondence with an initial segment of the natural numbers $0, 1, 2, ..., n$ (in which case the sequence is finite) or with the whole set of natural

numbers $0, 1, 2, ...$ (in which case the sequence is infinite). **Since a sequence may be finite, your functions will need to take this possibility into account.**

In this assignment, you will implement a module `Seq` for representing sequences. You should submit a single file, `Seq.hs` or `Seq.lhs`, that contains a module definition beginning with

```
module Seq
  where

type Seq a = [a]

-- function definitions start here.
```

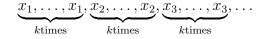All functions should be defined inside this module.

Problem 2.2.1. (15 pts)

As the above code suggests, you will implement sequences using infinite lists.

(a) Define a function `read ::  Seq a -> a -> (a, Seq a)` which takes a potentially infinite sequence as its first argument, and a default element as its second argument. If the sequence is non-empty, `read` should return the element at the front of the sequence paired with the rest of the sequence. If it is empty, `read` should return the default element paired with a representation of the empty sequence.

(b) Define a function `skip ::  Seq a -> (a -> Bool) -> Seq a` which takes a potentially infinite sequence along with a function of type `a -> Bool` and returns a sequence which skips all elements for which the function returns `False`. "Skipping" here means returning the next element in the sequence for which the function returns `True`.

(c) Define a function `twoseq ::  Seq a -> Seq a -> Seq a` which takes two sequences and returns a sequence which has every element in the first sequence, and if it is finite (it might not be), once the elements in the first sequence are exhausted, also has every element in the second sequence. You may not use `append`.

Problem 2.2.2. (25 pts)

(a) Define a *primitive recursive* function `dupk ::  a -> Int -> Seq a -> Seq a` which takes an element of some type `a`, an integer k, and a sequence. The function should place k copies of the element at the beginning of the sequence.

(b) Define a function `repeatk ::  Int -> Seq a -> Seq a` which takes a potentially infinite sequence $x_1, x_2, x_3, \ldots$ and returns a sequence of the form

$$\underbrace{x_1, \ldots, x_1}_{k\text{times}}, \underbrace{x_2, \ldots, x_2}_{k\text{times}}, \underbrace{x_3, \ldots, x_3}_{k\text{times}}, \ldots$$

In other words, the function should return a sequence where every element is duplicated k times. Points can be deducted for inefficient solutions.

(c) Define a function `addAdjacent ::  Num a => Seq a -> Seq a` which takes a sequence of integers $n_1, n_2, n_3, n_4, \ldots$ and returns an integer sequence of the form

$$n_1 + n_2, n_3 + n_4, n_5 + n_6, \ldots$$

In other words, the new sequence should consist of sums of adjacent elements in the old sequence.

(c) Define a function `addAdjacentk ::  Num a => Int -> Seq a -> Seq a` which is a generalization of the function from part (c). Given a sequence $n_1, n_2, n_3, n_4, \ldots$, it should return a sequence

$$n_1 + \ldots + n_k, n_{k+1} + \ldots + n_{k+k}, \ldots$$

Problem 2.2.3. (25 pts)[1]

(a) Define a function `binOpSeq ::  (a -> b -> c) -> Seq a -> Seq b -> Seq c` which takes a function `f` of two arguments along with two sequences $x_1, x_2, \ldots$ and $y_1, y_2, \ldots$, and returns a sequence consisting of the results of applying that function to each corresponding pair of elements:

$$(f \ x_1 y_1), (f \ x_2 y_2), (f \ x_3, y_3), \ldots$$

(b) Define functions `addSeq` and `mulSeq`, both of type `Num a => Seq a -> Seq a -> Seq a`, which add or multiply the corresponding elements of two sequences and return the sequence of results.

(c) Consider the definitions

```
ones :: Seq Int ones = 1:ones

foo :: Seq Int -> Seq Int foo xs = 1 : addSeq (foo xs) (foo xs)
```

and the Haskell expression
```
                        foo ones
```

What is the sequence to which this expression evaluates? To answer this question, implement a function `fooOnes ::  Seq Int` which generates this sequence using an *explicit* formula. (Hint: You may use `map` and list comprehensions).

(d) Consider the incomplete definition

---

[1]Parts (c) and (d) of this problem are based on a similar problem by Hongwei Xi.

```
facts :: Seq Int -> Seq Int facts xs = 1 : mulSeq (??) (??)
```

Replace the question marks with expressions so that `facts`, when applied to `[0..]` (the infinite ascending sequence of natural numbers), will return the sequence of factorial numbers (i.e. $0!, 1!, 2!, 3!, \ldots$).

Problem 2.2.4. (35 pts)

(a) Define a function `countTrue :: Seq Bool -> Seq Int` which takes a sequence of boolean values and replaces each element of the sequence with the count of how many times `True` appears in the sequence up to that point, e.g.

> `countTrue [True, False, True, True, False]` $\Longrightarrow$ `[1,1,2,3,3]`.

(b) Define a function `appFuns :: Seq (a -> a) -> Seq a -> Seq a` which takes a sequence of functions $f_1, f_2, f_3, \ldots$ of type `a -> a` along with a sequence of values $x_1, x_2, x_3, \ldots$ and returns a sequence in which the nth element is the result of applying functions f_1 through f_n to the element x_n, i.e.

$$f_1(x_1), f_2(f_1(x_2)), f_3(f_2(f_1(x_3))), f_4(f_3(f_2(f_1(x_4)))), \ldots$$

(c) Define a function `combos :: (a -> b -> c) -> Seq a -> Seq b -> Seq c` which takes a function `f: a -> b -> c` along with two sequences $x_1, x_2, x_3, \ldots$ and $y_1, y_2, y_3, \ldots$, and returns a sequence which contains the result of `f` applied to every single combination of elements from the two sequences, i.e.

> `f` $x_1 y_1$, `f` $x_1 y_2$, `f` $x_1 y_3$, $\ldots$, `f` $x_2 y_1$, `f` $x_2 y_2$, `f` $x_2 y_3$, $\ldots$

(c) Define a value `allPairs :: Seq (Int, Int)` which contains every single pair (n, m) of positive integers, including zero.

(d) Define a value `allAriths :: Seq [Char]` which contains every single string of the form "$5 * 6$" which is built from the digits 0 through 9 and one of the operators $*$, $+$, or $-$. You may want to use the sequence of strings representing the digits, `map show [0..9]`, the sequence of operators, `['+','-','*']`, and the function `(++)` to append strings.

2.3 The Shortest Superstring Problem

The shortest superstring problem is encountered in both DNA sequencing and data compression. The problem is defined in the following way: given a list of strings $S = \{s_1, \ldots, s_n\}$, what is the shortest possible string s^* such that every string $s_i \in S$ is a substring of s^*? In

this assignment, you will use higher-order functions to implement several algorithms for solving this problem. All your solutions should be defined within a module named `Superstring`, and you should submit the file `Superstring.hs` (or `Superstring.lhs`). **File names are case sensitive.** Verbal responses to non-programming questions should be in the form of comments in your code.

Note: All function definitions at top level must have explicit type annotations. Unless otherwise specified, your solutions may utilize functions from the standard prelude as well as material from the lecture notes and textbook. You may **not** use any other materials without explicit citation.

Problem 2.3.1. (25 pts)

All the algorithms you implement will be polymorphic. "Strings" will be represented as lists of elements of some type `a`. In your module, add the following type synonym declaration:

$$\text{type Str a = [a]}$$

Henceforth, whenever we talk about strings, we are referring to values of type `Str a`.

(a) Define a function `overlap ::  Eq a => (Str a, Str a) -> Int` which takes two strings s and s' and returns the number of elements at the end of s which overlap with the beginning of s'. For example:

```
overlap ("fire", "red")  =  2
overlap ("water", "blue")  =  0
```

You may want to use the function `isPrefixOf`, found in the prelude. To use it, you will need to add `import List` immediately after your module declaration.

(b) Define a function `contains ::  Eq a => Str a -> Str a -> Bool` which returns `True` only if the second string is a substring of the first string. Otherwise, it returns `False`.

(c) Define an infix operator `o ::  Eq a => Str a -> Str a -> Str a` which concatenates two strings s and s' by merging the overlapping elements at the end of s and the beginning of s'. For example:

```
"fire" ‘o‘ "red"  =  "fired"
"water" ‘o‘ "blue"  =  "waterblue"
```

(d) Using `foldr` and `(o)`, define a superstring algorithm `naive ::  Eq a => [Str a] -> Str a` that takes a list of strings, and constructs a superstring containing every string in that list.

Problem 2.3.2. (10 pts)

(a) Define a function `maximize ::  Eq a => (a -> Int) -> a -> a -> a` which takes a function f of type `a -> Int`, and two arguments of type `a`. It returns the argument which maximizes the function f.

(b) Define a function `minimize ::  Eq a => (a -> Int) -> a -> a -> a` which takes a function f of type `a -> Int` along with two arguments, and returns the argument which minimizes the function f.

Problem 2.3.3. (30 pts)

You will now define the optimal superstring algorithm.

(a) Using `filter`, define a function `update ::  Eq a => [Str a] -> (Str a,Str a) -> [Str a]` which takes a list of strings ℓ, and a pair of strings (s, s'). The function should combine the two strings s and s' (taking advantage of any overlapping) to obtain a new string s''. It should remove from the list of strings ℓ any strings contained in s'', and should then return a list which contains all of the remaining strings, as well as one copy of s''. For example:

```
update ["fire", "red", "blue"] ("fire", "red")  =  ["fired","blue"]
```

(b) Define a function `allPairs ::  Eq a => [Str a] -> [(Str a,Str a)]` which takes a list of strings, and returns a list of all pairs of strings which can be made out of this list, *excluding* any pairs of equal strings.

(c) Define a recursive algorithm `superstring ::  Eq a => ([Str a] -> [(Str a,Str a)]) -> [Str a] -> Str a` that takes as arguments a function *next* for generating pairs of strings out of a list, and a list of strings ℓ.

It should take the list of strings and generate a list of pairs using the supplied function. For *every* pair, it should generate a new list in which *only* that pair is combined, and should call itself recursively on this new, smaller list of strings (this can be done with either `map` or list comprehensions; remember that the recursive call is also expecting f as an argument). Once all the recursive calls have returned a result, it should choose the shortest result, and return it (you can use `foldr`, `length`, and `minimize` for this; for the base case of `foldr`, you can use `naive`). (Note: This can all be done in a single line, so plan carefully and use list functions wherever possible.)

Given an empty list, `superstring` should return an empty list. Given a list with only one string, it should return that string.

(d) Define a superstring algorithm `optimal ::  Eq a => [Str a] -> Str a` which takes a list of strings, and tries all possible ways of combining the strings, returning the best result. (Hint: All you need are your solutions from parts (b) and (c) above.)

Because it tries all possibilities, this algorithm always returns the shortest possible superstring, but runs in exponential time. Nevertheless, your implementation should be able to handle the following test cases:

```
test1  =  ["ctagcgacat", "aagatagtta", "gctactaaga", "gacatattgt", "tagttactag"]
test2  =  ["101001","010100010100", "100101", "001010", "11010", "100", "11010"]
test3  =  [x++y | x<-["aab","dcc","aaa"], y<-["dcc", "aab"]]
```

If you get a stack overflow on any of these examples, make sure that in `superstring`, recursive calls are made on progressively smaller lists, and that you are filtering out pairs of strings which are equal to each other.

Problem 2.3.4. (30 pts)

You will now define three greedy superstring algorithms. For all three of the functions below, you may assume that the list supplied as an argument will always have at least two elements.

(a) Define a function `firstPair ::  Eq a => [Str a] -> [(Str a,Str a)]` that takes a list of strings, and returns a list containing only one element: the first two strings as a tuple.

(b) Define a function `bestWithFirst ::  Eq a => [Str a] -> (Str a,Str a)` that takes a list of strings, and returns a list containing a single pair of strings. The first component in the pair should be the first string in the list. The second component in the pair should be the string in the list which has the largest overlap with the first string.

(c) Define a function `bestPair ::  Eq a => [Str a] -> (Str a,Str a)` that takes a list of strings, and returns a list containing the one pair which has the largest overlap.

By applying `superstring` to one of the the above functions, it is possible to produce one of three distinct greedy algorithms. Because each function produces a list with only a single element, these algorithms will not take exponential time.

(d) Define a superstring algorithm `greedy ::  Eq a => [Str a] -> Str a` which produces at least as short a superstring as any of the above three algorithms.

Problem 2.3.5. (5 pts)

One advantage of higher-order functions is that it is easy to write generic testing scripts.

(a) Define a function `compare ::  Eq a => ([Str a] -> Str a) -> ([Str a] -> Str a) -> [Str a] -> Double` which takes two superstring algorithms and a single list of strings, and returns the ratio between the length of the superstring generated by the first algorithm, and the length of the one generated by the second algorithm. You will need to use `fromIntegral` to convert to a `Double`.

(b) For the three test cases presented in **Problem #2.3.3**, how do the results of the four greedy algorithms compare to the optimal superstring?

Chapter 3

Algebraic Data Types and Type Classes

3.1 Assorted Practice Problems

In this assignment, you will implement a module `Data`. You should submit a single file, `data.hs` or `data.lhs`, that contains a module definition that begins with

```
module Data
  where

-- function definitions start here.
```

All data types and functions should be defined inside this module.

Problem 3.1.1. (25 pts)

(a) Define a new data type called `Color` which has four constructors, `Red::Color`, `Green::Color`, `Blue::Color`, and `Yellow::Color`, and is such that `show` and `(==)` are both defined for it. (Hint: This should all be done using a single `data` expression with a `deriving` clause added to it, consult [HCFP, Ch. 14, 16]).

(b) Define a new data type called `MyTree` which has two constructors, `MyLeaf::MyTree` and `MyNode::MyTree -> MyTree -> MyTree`, and is such that `show` and `(==)` are both defined for it.

(c) Define a new data type called `Natural` which has two constructors, `Zero::Natural`, `Succ::Natural -> Natural`, and is such that `show`, `(==)`, `(>)`, `(<)`, `(<=)`, and `(>=)` are all defined for it.

(d) Define a function `plusNat::Natural -> Natural -> Natural` which adds two naturals in the usual way.

(e) Define a function `nodeCount::MyTree -> Natural` which returns the number of nodes (including leaves) in a tree.

(f) Define a value `infTree::MyTree`, an infinite tree with no leaves.

(g) Now, define a polymorphic data type called `Tree a` which has two constructors, `Leaf::a -> Tree a` and `Node::a -> Tree a -> Tree a -> Tree a`.

(h) Define a value `infColorTree::Tree Color`, an infinite tree in which every node has a color, and the three neighbors of that node (the parent and its two children) have three distinct colors which are different from the color of the node. (That is, for every node v, the two children of that node must have different colors from each other as well as from v, and the parent of v must not share a color with either of the two children of v, nor with v itself).

Problem 3.1.2. (35 pts)

(a) Define a data type `Direction` with constructors `N`, `S`, `E`, and `W`, all of type `Direction`.

(b) Define a function `moveInDir::Direction -> (Int, Int) -> (Int, Int)` which takes a direction along with a coordinate (x, y) on the Cartesian plane and returns the coordinate after 1 step is taken in the appropriate direction (up is North, left is West).

(c) Define a data type `Path` with constructor `Path :: [Direction] -> Path`.

(d) Define a function `isLoop::Path -> Bool` which takes a path and returns `True` if and only if following the directions in the path will bring you back to the point at which you start.

(e) Define a data type `Itinerary` with constructors `Walk::Path -> Itinerary -> Itinerary`, `Wait::Natural -> Itinerary -> Itinerary`, and `Stop::Itinerary`, and a data type `Outcome` with constructors `Win::Outcome`, `Lose::Outcome`, `Tie::Outcome`, and `Uncertain::Outcome`.

(f) Implement a function `race::Itinerary -> Itinerary -> Outcome`. This function should return `Uncertain` if the two itineraries don't end up in the same place when followed from the same starting point, `Win` if the first itinerary stops at the endpoint first, `Lose` otherwise, and `Tie` if they stop at the same time. Time is measured using naturals, and moving n steps takes the same amount of time as waiting n.

Problem 3.1.3. (40 pts)

In this problem, a "constraint" is a function which determines whether an element of some type `a` satisfies some property, and returns `True` if only if it does.

(a) Define a polymorphic data type `Constraint a` with a single constructor `Con::(a ->
Bool) -> Constraint a`. Note that `Con` acts as a wrapper for functions, so a value
of type `Constraint a` cannot be applied directly to an argument, but must first be
unwrapped using a pattern match.

(b) Define a function `andCon::Constraint a -> Constraint a -> Constraint a` which
takes two constraints and returns a third constraint that is satisfied only if the two
component constraints are satisfied.

(c) Define a function `listCon::[Constraint a] -> Constraint a` which takes a list of
constraints and returns a constraint that is satisfied only if all the constraints in the
list are satisfied.

(d) Implement a function `findSol::Ord a => Constraint a -> [a] -> Maybe a` which
takes a constraint as its first argument, a list of possible solutions to try as its second
argument, and returns the *smallest* (as determined by `(<)` or `min`) solution found in
the list which satisfies the constraint. If no solution is found, the function should return
the value `Nothing`.

We will define equations of two variables as constraints. Include the following definition in
your code.

```
type Equation = Constraint (Int, Int)
```

(d) Using `findSol`, define a function `findIsect::[Equation] -> (Int, Int) -> (Int,
Int) -> Maybe (Int, Int)` which takes a list of equations to satisfy, along with two
points in the Cartesian plane. These points indicate the bottom-left and upper-right
corners of the rectangle in which to look for integer solutions that satisfy *all* the equa-
tions simultaneously. The function should return the smallest (as defined by `(<)` or
`min`) solution to the equations, if any exists. (Hint: Use a comprehension to generate
a list of potential solutions).

(e) Define a value `solution::Maybe (Int, Int)` which is the least solution to the system
of equations

$$
\begin{aligned}
y &= (3-x)^4 - (3-x)^3 - (3-x)^2 + 5 \\
y &= x + 2.
\end{aligned}
$$

All the solutions can be found between $(0,0)$ and $(10,10)$. (Hint: use lambda expres-
sions and `(==)` to write an equation as a constraint, e.g. $y = 3x$ can be written as `\(x,
y) -> y == 3 * x`).

3.2   Auction Algorithms

An auction is an algorithm that takes a collection of bids for bundles of one or more items, and determines which of those bids will be satisfied (that is, "win"). A payment scheme determines how much each winning bidder will pay. In this assignment, you will use data types and type classes to implement an auction algorithm and a few payment schemes. All your solutions should be defined within a module named `Auction`, and you should submit the file `Auction.hs` (or `Auction.lhs`). **File names are case sensitive.**

Note: All function definitions at top level must have explicit type annotations. Unless otherwise specified, your solutions may utilize functions from the standard prelude as well as material from the lecture notes and textbook. You may **not** use any other materials without explicit citation.

Problem 3.2.1. (20 pts)

(a) In an auction, bidders need a way to represent their bid for one or more items. An auction bid consists of a price the bidder is willing to pay,

```
type Price = Int
```

and a description of the items the bidder wishes to buy, which could be of any type `a`. Define a new data type called `Bid a` which has a single constructor, `Bid::Price -> a -> Bid a`, and is such that `show` and `(==)` are both defined for it.

You will define several data types for representing descriptions of items in three different auction settings. Each will be defined as a member of the following class:

```
class Bundle a where
  conflict :: Eq a => a -> a -> Bool
```

Add this declaration to the beginning of your module. For any type in the `Bundle` class, you must define a function `conflict` which returns `True` if the first argument is a bundle that overlaps in any way with the second, and `False` otherwise.

(b) Suppose that an auction offers only a single item for purchase. Define a new data type called `Single` which has one constructor, `Item::Single`, and is such that `show` and `(==)` are both defined for it. Write an `instance` declaration so that `Single` is in the class `Bundle`. Because there is only one value of the type `Single`, the bundle it represents always conflicts with itself.

(c) Suppose that an auction has an arbitrary number of uniquely labelled items for sale, and that each bidder can bid for only one item. Let `Label` be a type synonym for `Int`:

```
type Label = Int
```

Define a new data type called `Labelled` which has a single constructor, `Labelled::Label -> Labelled`, and is such that `show` and `(==)` are both defined for it. Write an `instance` declaration so that `Labelled` is in the class `Bundle`. A single-item bundle only conflicts with another single-item bundle if the items in both bundles have the same label.

(d) Suppose an auction has an arbitrary number of uniquely labelled items, and the bidder is allowed to bid for any particular set of these items. Define a new data type called `Set` which has a single constructor, `Set::[Label] -> Set`, and is such that `show` and `(==)` are both defined for it. Write an `instance` declaration so that `Set` is in the class `Bundle`. In this case, two bundles conflict only if they share at least one item with the same label.

(e) To make testing and solving subsequent problems easier, define the functions `bids::[Price] -> [Bid Single]` and `bids'::[(Price,Label)] -> [Bid Labelled]`. This makes it possible to generate many test cases using list comprehensions.

Problem 3.2.2. (30 pts)

An allocation takes a list of bids and returns a list of outcomes indicating which bids won and which bids lost:

```
type Allocation a = [Bid a] -> [Outcome a]
```

(a) Define a data type `Outcome a` with two constructors `Win::Price -> a -> Outcome a` and `Lose::Outcome a`. Make sure that `show` and `(==)` are both defined for it.

(b) Define a function `minPrice::(Eq a,Bundle a) => a -> [Outcome a] -> Price` which takes a bundle description x of type `a` and a list of outcomes, and returns the minimum bid price necessary to win that item given the outcomes. If x conflicts with multiple winning bids, the minimum price must exceed the *total* of the prices offered for all conflicting items. If x conflicts with no winning bids, the minimum price is 0.

(c) Define a function `addBid::(Eq a,Bundle a) => Bid a -> [Outcome a] -> [Outcome a]` which takes a bid `bid` and a list of outcomes and adds an outcome to the head of the list which indicates whether `bid` wins or loses. The bid wins *only* if the price offered by the bid is strictly greater than the minimum necessary price to win the bundle specified by the bid.

(d) Define an allocation algorithm `alloc::(Eq a,Bundle a) => Allocation a` which takes a list of bids and determines (using `addBid`) which bids should win, starting with the right-most bid (the one at the end of the list) and moving left.

(e) The "welfare" of an allocation is defined as the total of the prices offered by all the winning bidders. Define a function `welfare::(Eq a,Bundle a) => [Bid a] -> Price` which takes a list of bids and computes the welfare for `alloc`.

Problem 3.2.3. (30 pts)

For any auction, it is possible to define many payment schemes. A payment scheme is used to determine how much the winning bidders actually pay. We define the following type synonym for payment schemes:

$$\texttt{type Payment a = [Bid a] -> [Price]}$$

A payment scheme takes an auction and list of bids, and returns a list containing the price each of the bidders in the input list must pay. Bidders that lose always pay 0. All the payment schemes defined in this problem are for the `alloc` algorithm defined in the previous problem.

(a) Define a payment scheme `payBid::(Eq a,Bundle a) => Payment a` that requires that every winning bidder pay the price they offered in their bid.

(b) Define a function `isolateEach::[a] -> [([a],a,[a])]` that takes a list of elements, and returns a list of tuples. In each tuple, exactly one of the elements is isolated from the rest of the list. For example:

`isolateEach [1,2,3,4]  =  [([],1,[2,3,4]),([1],2,[3,4]),([1,2],3,[4]),([1,2,3],4,[])]`

(c) Define a payment scheme `payMin::(Eq a,Bundle a) => Payment a` under which each winning bidder must pay the minimum amount necessary for it to win (this is *not* the bidder's bid price, it can be lower). Look carefully at how `alloc` works to determine how to compute this value. If there is only a single item, this payment scheme corresponds to the second-price Vickrey auction.

(d) Define a payment scheme `payDiff::(Eq a,Bundle a) => Payment a`, in which each bidder must pay an amount equal to the the welfare of the auction if the bidder is present minus the welfare of the auction if the bidder is removed. You will need to run two separate auctions for each bidder to determine these values.

(e) The revenue of an auction is defined to be the sum of the payments made by the winning bidders under the payment scheme. Define a function `revenue::(Eq a,Bundle a) => Payment a -> [Bid a] -> Price` which computes the revenue for `alloc` given a payment scheme and list of bids.

Problem 3.2.4. (20 pts)

Solve each of the problems below by computing and comparing the revenues over all the possible combinations.

(a) Consider a situation in which there is exactly one item, and each bidder can supply a bid in the range `[1..10]` for this item. Suppose there are exactly three bidders. If every possible combination of bids is equally likely and you want to maximize revenue, would you rather use the `payDiff` or `payMin` payment scheme? Define a function `partA::Bool` which is `True` if `payDiff` is preferred, and `False` if `payMin` is preferred.

(b) Consider a situation in which there are four labelled items, and each bidder can supply
    a bid in the range `[1..4]` for any one of these items. Suppose there are again exactly
    three bidders. If every possible combination of bids is equally likely and you want
    to maximize revenue, would you rather use the `payDiff` or `payMin` payment scheme?
    Define a function `partB::Bool` which is `True` if `payDiff` is preferred, and `False` if
    `payMin` is preferred.

It should be clear that revenue is always highest under the `payBid` scheme. However, in an
auction in which bidders must pay the exact bid they offer, rational bidders will tend to
misrepresent their true value by underbidding. Rational bidders are less likely to do this
under the other two payment schemes. To model this, we can divide the revenue under a
`payBid` scheme by a constant factor `k::Int`. Use integer division for these problems.

(c) For the scenario described in part (b) above, how large must `k` be to ensure that `payMin`
    is preferred over `payBid`? Define a function `partC::Int` which computes this value.

(d) For the scenario described in part (b) above, how large must `k` be to ensure that `payDiff`
    is preferred over `payBid`? Define a function `partD::Int` which computes this value.

Chapter 4

Abstract Data Types and Proofs

4.1 Proofs as Data Values

In this assignment, you will implement three modules, `Function`, `Proof`, and `Natural`. You should submit three files, `Function.hs`, `Proof.hs`, and `Natural.hs` (or *.lhs).

Problem 4.1.1. (10 pts)

You will implement a module `Function`, and you should submit the file `function.hs` or `function.lhs`, which contains a module definition that begins with:

```
module Function where
-- definitions start here.
```

(a) Write a `class` definition for a class `Function`. In order for a type `p` to be a member of this class, there must exist two functions, `app::Eq a => p a b -> a -> b` and `compose::(Eq a, Eq b) => p b c -> p a b -> p a c`.

(b) Define a polymorphic data type `DispFun a b` which has a single constructor, `DispFun::String -> (a -> b) -> DispFun a b`. Write an instance declaration so that any type `DispFun a b` is in the class `Show`.

(c) Implement a function `dispplus::DispFun (Int, Int) Int` which adds two `Int`s.

(d) Implement a function `appDispFun::DispFun a b -> a -> b` which takes a displayable function and applies it to an argument, returning the result.

(e) Implement a function `composeDispFun::DispFun b c -> DispFun a b -> DispFun a c` which composes two functions in the usual way, but also composes the strings representing them. A reasonable string representation of composed functions will suffice.

(f) Write an instance declaration which makes `DispFun` a member of the `Function` class.

Problem 4.1.2. (15 pts)

Solutions to this problem should also occur in the module `Function`.

(a) Define a polymorphic data type `FiniteFun a b` which has a single constructor, `FiniteFun::[(a, b)] -> b -> FiniteFun a b`. A value of type `FiniteFun a b` represents a function which for a finite number of values of type `a` has a defined result of type `b` (this is represented by the list of pairs), and for all other values, always returns a default value of type `b` (the second argument to the constructor).

(b) Implement a function `appFiniteFun::Eq a => FiniteFun a b -> a -> b` which takes a finite function and applies it to an argument, returning the result.

(c) Implement a function `composeFiniteFun::(Eq a, Eq b) => FiniteFun b c -> FiniteFun a b -> FiniteFun a c` which returns a `FiniteFun a c` value which acts like the composition of the two functions supplied as arguments.

(d) Write an instance declaration which makes `FiniteFun` a member of the `Function` class.

Problem 4.1.3. (25 pts)

You will implement a module `Proof`, and you should submit the file, `proof.hs` or `proof.lhs`. Your module definition should export only the following types and values: `Pf`, `refl`, `symm`, `tran`, `axiom`, and `appPf`. No value constructors should be exported from the module.

(a) Define a polymorphic data type `Pf a` with a constructor `Equal::a -> a -> Pf a`.

(b) Create an `instance` definition which says that if a type `a` is in the class `Show`, the type `Pf a` is also in the class `Show`. You will need to supply a `show` function which returns a `String` given a `Pf a`. One convenient way to display a proof is to first show the left-hand side of the equation, then an equals sign, and finally the right-hand side of the equation.

(c) Implement a function `refl::a -> Pf a` which returns a trivial "proof" (that is, a value of type `Pf a`) that the argument is equal to itself.

(d) Implement a function `symm::Pf a -> Pf a` which switches the two sides of the proof with one another (i.e. a proof of $x = y$ becomes $y = x$).

(e) Implement a function `tran::Eq a => Pf a -> Pf a -> Pf a` which takes two proofs, one which represents that $x = y$, and one which represents that $y' = z$, and first checks whether $y = y'$ (use (`==`) to check this). If y does equal y', the function should return a proof that $x = z$. If not, it should simply return $x = y$ unchanged.

(f) Implement a function `axiom::a -> a -> Pf a` which acts as a wrapper for the `Equal` constructor.

(g) Implement a function `appPf::(a -> a) -> Pf a -> Pf a` which applies a function `f::a -> a` to both sides of a proof. (Extra Credit: Comment on how the `appPf` function actually breaks the modularity of the proofs being built. What's an example of an argument to `appPf` which might break a proof?)

Problem 4.1.4. (50 pts)

In this problem, you will prove that for all natural numbers n represented using the `Natural` data type, it is the case that $0 + n = n + 0$. You will do this by writing a program which, for a given n, generates such a proof.

Note: This problem is challenging. If anything is unclear, don't hesitate to ask questions.

You will implement a module `Natural`, and you should submit the file, `natural.hs` or `natural.lhs`. You'll want to have an `import` statement in your module definition, so that you are able to build proofs:

```
module Natural where
import Proof

-- function definitions start here.
```

Note that for this problem, you may not use any constructors which were not explicitly exposed in the definition of the `Proof` module. Particularly, you may *not* use pattern matching over the `Equal::a -> a -> Pf a` constructor.

You should use the following definition for `Natural`:

```
data Natural = Zero | Succ Natural | Plus Natural Natural
  deriving (Show, Eq)
```

(a) One possible inductive definition for addition on natural numbers might look something like this:

```
Plus n Zero     = n
Plus n (Succ m) = Succ (Plus n m)
```

The way to read the above definitions as logical statement is, "for all `Natural` numbers n, (Plus n Zero) is equal to n," and, "for all `Natural` numbers n and m, (Plus n (Succ m)) is equal to (Succ (Plus n m))."

Write two functions, `basPlus::Natural -> Pf Natural` and `indPlus::Natural -> Natural -> Pf Natural`. These should encode the two parts of the definition of `Plus`

as axioms. For example, given the natural `Zero::Natural`, `basPlus Zero` should return a value of type `Pf Natural` which states that `(Plus Zero Zero)` is equal to `Zero`.

(b) Implement a function `succPf::Pf Natural -> Pf Natural` which applies `Succ` to both sides of a proof.

(c) Implement a function `zeroCommPf::Natural -> Pf Natural` which returns a proof that for a natural `n::Natural` (the supplied argument), it is the case that `(Plus Zero n)` is equal to `n`.

As an example, supplying 2 (that is, `Succ (Succ Zero)`) to `zeroCommPf` should yield the proof that $0 + 2 = 2$:

$$\text{zeroCommPf (Succ (Succ Zero))}$$
$$\Downarrow$$
$$\text{Equal (Plus Zero (Succ (Succ Zero))) (Succ (Succ Zero))}$$

Note that this is *different* from the definition of `Plus`, in that `n` and `Zero` are reversed.

(Hints: It is strongly recommended that you first try to prove this by hand. For the base case, build a proof that `(Plus Zero Zero)` is equal to `Zero`. For the inductive case, build up an inductive hypothesis by making a recursive call, then use `succPf`. What does this give you? How can the `indPlus` function help? Use this along with `tran::Pf a -> Pf a -> Pf a` to build up the inductive step. You can use `symm` with `tran` to combine proofs in various ways.)

(d) Implement a function `pf::Natural -> Pf Natural` which generates a proof that `(Plus n Zero)` is equal to `(Plus Zero n)` for any supplied argument `n::Natural`. (Hint: You need to combine the result from part (c) with one of the axioms representing the definition of `Plus`). Another sample input and output:

$$\text{pf (Succ (Succ Zero))}$$
$$\Downarrow$$
$$\text{Equal (Plus Zero (Succ (Succ Zero))) (Plus (Succ (Succ Zero)) Zero)}$$

Chapter 5

An Interpreter for mini-Haskell

The following problems relate to the implementation of an interpreter and type checker for a subset of Haskell.

5.1 Call-by-value Interpreter

In this assignment, you will implement a call-by-value interpreter for a small subset of Haskell. You will make changes to the `Env` and `Eval` modules.

Included with the code is a parser library, *Parsec*, necessary to compile and use the interpreter. When using Hugs, it should be sufficient to load the `Main` module, and to run `mainParseEval "tests1.mhs"`. The files should run as provided, so let us know if you run into problems.

Familiarize yourselves with the `Exp` data structure in the `Exp` module (`Exp.hs`), which represents the abstract syntax of the language. The `Val` data structure, found in the `Val` module (`Val.hs`), represents evaluated (a.k.a. normalized) expressions in the language. Note that the two correspond in the base cases (integers, booleans, unit, empty list, built-in operators, and lambda abstractions).

Note that you are not allowed to modify any of the existing data type definitions unless a problem states otherwise or explicitly directs you to change a definition. There are, however, crude `show` functions for expressions, and you may want to use them at times when debugging (think about returning a subexpressions as a string in an error message).

Problem 5.1.1. (30 pts)

In this problem, you will implement an interpreter for expressions without variables, abstractions, or let-bindings. Because the parsed code is not being type checked, errors may occur during evaluation, so you will need to use `Error Val` as the return type for the various evaluation functions.

(a) In the `Eval` module, implement the body of the function `appOp ::  Oper -> Val ->`
    `Error Val`, which takes an operator and a value, and when the operator is defined
    on that value in mathematical terms, returns the result (unary operators include `Not`,
    `Head`, and `Tail`.

    If the operator is binary, take advantage of the `Partial::Oper -> Val -> Val` con-
    structor, which can represent a partially applied built-in function. For example, `appOp`
    might return a value like `Partial Plus (N 4)` if it must apply the binary operator
    `Plus` to the single value `N 4`.

    If the operator is not defined on that value, the function should return an error. Be
    careful with list operators such as `Head`, as they are not defined on empty lists. When
    in doubt, you can use the real Haskell interpreter to determine how this function should
    behave.

(b) In the `Eval` module, implement the body of the `appBinOp::Oper -> Val -> Val ->`
    `Error Val` function, which takes an operator and two values, and returns the resulting
    value if it is defined. Otherwise, it returns an error. Don't forget that `Equal` is defined
    on both numbers and booleans. You are not required to implement equality on lists,
    but you may do so for a small amount of extra credit.

(c) In the `Eval` module, implement the body of the `appVals::Val -> Val -> Error Val`
    function, which takes a pair of values where the first value is either a unary operator,
    a binary operator, or a partially applied binary operator. You should not need to use
    any functions other than those you defined in parts (a) and (b).

(d) In the `Eval` module, implement the body of the `ev0::Exp -> Error Val` function,
    which evaluates expressions which do not contain let-bindings, lambda abstractions, or
    variables. Thus, it should evaluate all base cases (such as operators, booleans, integers,
    unit, and the empty list) as well as `if` statements and applications. For all other cases,
    `ev0` may return an error.

    You *should not* evaluate both branches of an `if` statement, and this is tested in the file
    `tests1.mhs`.

You should now be able to test the interpreter on some input. When using Hugs, it should
be sufficient to load the `Main` module, and to run `mainParseEval "tests1.mhs"`. You may,
of course, write and try evaluating your own test code.

Problem 5.1.2. (20 pts)

In the module `Env`, you will implement an environment data structure which can be used to
store the values (and in later assignments, types) of variables.

(a) We will represent variable names with values of type `String`. Choose a representation
    for environments by choosing a definition for the polymorphic type `Env a`, which is

used to store relationships between variable names and values of type `a`. Make sure the type name is exported from the module, as it will be needed in the `Eval` module.

(b) Define a value `emptyEnv::Env a` which will represent the empty environment (no variables), and make sure it is exported from the `Env` module.

(c) Define a function `updEnv::String -> a -> Env a -> Env a` which updates the environment with a new association between a variable and a value. If an association already exists in the environment, it should ***not*** be removed or overwritten. The new association should, however, hide any previous associations from being found. Make sure this function is exported from the module.

(d) Define a function `findEnv::String -> Env a -> Maybe a` which takes a variable name and retrieves the value associated with that variable, returning `Nothing` if there exists no such association in the environment. If in a given environment, multiple associations exist with the variable name in question, this function should always return the associated value which was inserted most recently. Export this function from the module.

Problem 5.1.3. (50 pts)

You will now implement a full interpreter for the language.

(a) In the `Eval` module, add two more cases to the body of the `appVals::Val -> Val -> Error Val` function, one for lambda abstractions with variables, and one for lambda abstractions with unit. You will need to call the `ev::Exp -> Env Val -> Error Val` function, which you have not yet implemented.

Note that a lambda abstraction with unit can only be applied to the unit value. Note also that a lambda abstraction is represented at the value level as a *closure* – it holds a copy of the environment under which the abstraction occurred in the program. The expression inside the lambda abstraction should be evaluated under this stored environment. For non-unit lambda abstraction, you should remember to extend this environment, however, by associating the formal variable in the abstraction with the argument value.

(b) Implement the body of the `ev::Exp -> Env Val -> Error Val` function, which evaluates all expressions which do not produce an error, and returns an error otherwise.

The base cases should be similar to those of `ev0`, except for the case of a variable. A variable evaluates to the value with which it is associated in the environment. If it is not found in the environment, it is not bound, and an error should be returned. When encountering lambda abstractions, remember to store the current environment inside them.

Remember that this interpreter is call-by-value, so `let` bindings and arguments should be evaluated eagerly. And, as before in `ev0`, you *should not* evaluate both branches of an `if` statement.

(c) Modify the `evalExp::Exp -> Error Val` function in the `Eval` module to call `ev` instead of `ev0`.

You should now be able to test your interpreter. The file `tests2.mhs` contains a program which outputs the prime numbers between 2 and 20. You are encouraged to write additional test cases, and particularly insightful or revealing test cases may receive some extra credit.

5.2   Call-by-name Interpreter

In this assignment, you will implement a call-by-name interpreter for a small subset of Haskell. You will continue to modify the Haskell skeleton code from Homework Assignment #6. You may use the Homework Assignment #6 solutions, either those posted or your own, as a starting point. You will make changes to the `Exp` and `Eval` modules.

Note that you are still not allowed to modify any of the existing data type definitions unless a problem states otherwise or explicitly directs you to change a definition.

Note: Problem 4 is a bonus problem, a solution for which is worth up to 25 points of extra credit.

Problem 5.2.1. (30 pts)

(a) In the `Exp` module, implement a recursive function `noLets::Exp -> Exp` which takes an expression as an argument, and converts all `let` subexpressions found in that expression into applications of lambda abstractions (see Figure 5 in Section 3 of Handout 27).

(b) In the `Exp` module, implement a function `subst::String -> Exp -> Exp` which takes a variable name x, an expression N for which that variable must be substituted, and finally, an expression M on which to perform the substitution.

The function can perform the substitution naively (that is, you may assume that there are no variable name collisions), but it must not substitute any bound variables, even if they have the same name. Think carefully about the checks which need to be performed when `subst` encounters a lambda abstraction.

If throughout the assignment you never call `subst` on an expression which may have a `let` binding as a subexpression, you do not need to include a case for `let` bindings in your definition of `subst`.

Problem 5.2.2. (45 pts)

(a) In the `Eval` module, implement a function `appValExp::Val -> Exp -> Error Val` which evaluates a value applied to an expression. You should not evaluate the second arguments of the short-circuited boolean binary operators (i.e. `((&&) False)` and

`((||) True)`), nor the argument passed to a unit lambda (that is, `\() -> ...`) abstraction. Note that for some cases, `appValExp` will need to call `subst`, as well as `ev0`, which you will implement in part (b). For convenience, you may call `appVals` from the previous assignment, but do this very carefully. You should not evaluate any subexpression which does not need to be evaluated according to the evaluation rules.

(b) In the `Eval` module, modify the body of the `ev0::Exp -> Error Val` function so that it evaluates *all possible* expressions according to the call-by-name evaluation *substitution* model, as defined in Section 3 of Handout 27. If you apply `noLets` to an expression before calling `ev0` (for example, by modifying the wrapper `evalExp::Exp -> Error Val`), you can be certain that the only situation in which you will need to perform a substitution is at an application of a lambda abstraction, which is a case already handled by `appValExp`.

Problem 5.2.3. (25 pts)

(a) In the `Eval` module, modify the body of the `ev::Exp -> Env Val -> Error Val` function so that it evaluates expressions according to the call-by-name evaluation *environment* model, as defined in Section 4 of Handout 27. You may use the `subst` function to replace variables with thunks applied to unit. In your solution, you may simply reuse the variable being substituted as the variable bound to the thunk.

(b) Create a file named `tests3.mhs`, and in it, write a small program on which the call-by-value interpreter implemented in the last assignment would diverge, but on which the call-by-name interpreter implemented in this assignment converges.

Problem 5.2.4. (*25 extra credit pts)

You will implement a transformation on expressions which ensures that the implementation of `subst` in Problem 1(b) works correctly for all programs.

(a) In the `Exp` module, implement a recursive function `unique::[String] -> Env String -> Exp -> (Exp, [String])` which takes an expression and gives a unique name to every variable in that expression, ensuring that no two lambda abstractions are over variables with identical names. You will need to maintain a list of fresh variable names. This list must also be returned with the result because an expression may have multiple branches, and variables cannot repeat even across branches. The environment is used to associate old variable names with new ones. You may use `noLets`.

(b) Modify `evalExp::Exp -> Error Val` so that an expression is evaluated only after being transformed using `unique`.

5.3   Type Inference

In this assignment, you will implement a type inference algorithm for a small subset of Haskell. You will make changes to the `Ty` module. The only modules which are different from previous versions of the interpreter are the `Main` and `Ty` modules, so you may swap in either your own versions of the `Eval`, `Exp`, and `Env` modules, or the versions provided as solutions for Homework Assignments #6 and #7.

You may not modify any of the existing data type definitions unless a problem states otherwise or explicitly directs you to change a definition.

Note: Problem 5 is a bonus problem, a solution for which is worth up to 30 points of extra credit.

Problem 5.3.1. (25 pts)

You will implement a type checker for a subset of mini-Haskell. Familiarize yourself with the abstract syntax for types (corresponding to Section 2 of Handout 23). Note that there are base types, and a constructor `Arrow::Ty -> Ty -> Ty` to construct a function type.

  (a) Implement a function `tyOp::Oper -> Ty` which returns the type of an operator. You may assume that `(==)` can only be applied to integers, and that only integer lists can be constructed (which means, for example, that the type of `[]` is an integer list).

  (b) Implement a function `ty0::Exp -> Error Ty` which can successfully type check all primitives (unit, empty list, operator, integer, and boolean), `if` statements, applications, and unit lambda abstractions (these are all the cases which do not have variables).

   Since there are no type variables, you may use derived equality (`==`) on types. Remember that both branches of an `if` statement must have the same type, and that an `if` condition must have a boolean type. For application, you may need to pattern match on the type of the function in order to check that its type matches the type of its argument.

   Your solution should be able to type check `tests4.mhs` successfully, and should reject `tests1.mhs` (provided with previous assignments) for not being well-typed.

Problem 5.3.2. (10 pts)

Performing type inference on expressions which may contain variables requires *type* variables (in terms of the syntax for types, it is a type part of which is built using the `TyVar::String -> Ty` constructor). We cannot simply use derived equality on types when type variables are present, so we will need to define a unification algorithm which attempts to find a minimal substitution of type variables which makes two types equal. Informally, two types which contain variables are equal if we can somehow replace those variables in both expressions and obtain two equal expressions. For example, given the types `Int -> b` and `a -> Bool`,

the minimal substitution which makes them equal is one which replaces all occurrences of `a` with `Int` and all occurrences of `b` with `Bool`.

In order to implement this algorithm, we will need to define substitutions, which are simply functions that substitute occurrences of a type variable with some other type. Thus, the type `Subst` is defined inside the `Ty` module to be

```
type Subst = Ty -> Ty
```

You will define a few functions for constructing and manipulating substitutions.

(a) Define a function `idsubst::Subst` which makes no changes to a type, and a function `o: Subst -> Subst -> Subst` which takes two substitutions and applies them in sequence, one after another.

(b) Define a function `subst::String -> Ty -> Subst` which takes a string x representing a type variable name, and a type t which will be used to substitute that variable. It should then return a substitution which, when given a type t', will substitute all occurrences of x in that type with t. Note that it *should not* substitute any other type variables which might be found in the type t'.

(c) Notice the type definition `type FreshVars = [Ty]`. Define a value `freshTyVars::FreshVars`, an infinite list of type variables in which no type variable is ever repeated.

Problem 5.3.3. (25 pts)

You will implement the unification algorithm for types. We say a substitution `s` *unifies* two types `t1` and `t2` if `s t1 == s t2`, where `==` is derived equality on types.

(a) Define a function `unify::Ty -> Ty -> Error Subst` which takes two types and finds the minimal substitution which makes them equal if one exists, and returns an error otherwise.

Note that given two types with no type variables, as long as they are equivalent in terms of derivable equality, a trivial substitution which makes no changes to the types is sufficient to unify them. On the other hand, if two types without type variables are not equivalent, there exists no substitution which can unify them.

Also, remember that a type variable can be substituted for any type, including another type variable. Finally, be very careful when unifying two function types (constructed using `Arrow::Ty -> Ty -> Ty`. First, try to unify the argument types. If a substitution is obtained, be sure to apply it to the result types *before* trying to unify them. For example, consider the two types `a -> a` and `Int -> Bool`, which cannot be unified. If we try to unify `Int` and `a`, we will obtain a substitution which replaces all instances of `a` with `Int`. This should be taken into account when trying to unify `Bool` and `a`.

Problem 5.3.4. (40 pts)

You will complete the already partially-implemented type inference function `ty::Env Ty -> FreshVars -> Exp -> Error (Ty, Subst, FreshVars)`. This function maintains an environment which maps variables in expressions to their types, and binds fresh *type* variables to any new variables it encounters in lambda abstractions or let-bindings. Each time the function processes some node in an expression tree, it may need to unify the types of the children of that node, and so, it accumulates these substitutions, returning the to the caller.

Note carefully the base cases for primitives, which return the trivial substitution, along with the types

  (a) Add the base cases for expressions (unit, the empty list, integers, boolean values, and operators) to the definition of the function `ty`. You may simply return the fresh list of variables unchanged, and a trivial substitution which makes no changes to the types.

  (b) The case for `if` expressions in the definition of `ty` is not complete. Notice that the function

```
    tys::Env Ty -> FreshVars -> [Exp] -> Error ([Ty], Subst, FreshVars)
```

   is used to obtain the types for all three subexpressions simultaneously. Perform the remaining checks and unifications in order to complete this part of the definition of the function. Remember that any substitutions you might generate in the process (including the one that has already been generated when type checking the three subexpressions) need to be considered in every subsequent unification. Also, when returning the substitution(s) you generated, remember to use `o::Subst -> Subst -> Subst` to combine them in the proper order.

  (c) The case for unit lambda abstractions in the definition of `ty` is not complete. Complete this part of the definition. Remember to return the type of the abstraction (it takes a value of type unit as an argument).

  (d) The case for general lambda abstractions in the definition of `ty` is not complete. Complete this part of the definition. Because a new variable is encountered, you will need to obtain a fresh *type* variable, and bind this expression variable to the fresh type variable in the environment. The subexpression should be type-checked under this new environment.

   Remember that the argument type of the abstraction is the same as the type of the variable over which the abstraction is defined. You may want to look at the part of the definition of `ty` for let bindings for some guidance. The cases for variables and application have already been completed for you.

Problem 5.3.5. (*30 extra credit pts)

(a) Notice that the mini-Haskell type inference algorithm is able to generate polymorphic types for individual values. For example, given the expression `\ x -> x`, the type should look something like `t1 -> t1`, where `t1` is a fresh variable generated using `freshTyVars`. Define the function `freevars::Ty -> [String]`, which returns the list of free type variables in a type.

(b) In the `Ty` module, notice the definition of a syntax for polymorphic types, `data PolyTy = ForAll [String] Ty`. Use your solution from part (a) to implement the function `canon::Ty -> PolyTy` which takes a type $t(\alpha_1, \ldots, \alpha_n)$ and returns a polymorphic type $\forall \alpha_1 \ldots \alpha_n . t(\alpha_1, \ldots, \alpha_n)$ in which every free variable has been universally quantified.

(c) The `Main` module uses values of type `data AnnotVal = AnnotVal Val Ty` to display values annotated with their types. Modify the `show` function for `AnnotVal` so that if a value's type is polymorphic, it is displayed in its explicitly quantified form.

(d) In Problems #1 and #3, we assumed that `(==)` can work only on integers. Modify the function(s) `ty0` and/or `ty` (as well as any others you may need to modify, such as `tyOp`) so that `(==)` will type check if it is applied to two boolean values as well as to two integers.

(e) Try using a similar approach to make `[]` and `cons` polymorphic. (Note: This problem is fairly open-ended and potentially challenging).

Bibliography

[HCFP] Thompson, Simon. *Haskell: The Craft of Functional Programming,* Second Edition, Addison-Wesley, paperback, 1998.