Dennis George
CS360 Test 2
8/28/2020
Start Time: 1:00PM
End Time: 3:00PM
Attempted Problems: 1, 2, 3, 5, 7

1.

(i) p, q are relatively prime, i.e. they do not have common factors except for 1

$RPRIME(P, Q) \Leftrightarrow (\forall I > 1, I \varepsilon N), (NOT(\ (\exists I) st\ (P|I\ AND\ Q|I\ )))$

Numbers P and Q are relatively prime if and only if there exists a natural number I which is greater than 1 and it is not the case that there exists an I which divides both P and Q.

(ii) p, q are different and they have exactly one non-trivial common factor, i.e. different from 1

$P \neq Q\ AND\ |\{(P|I\ AND\ Q|I\ )\ |\ (\forall I > 1, I \varepsilon N\ )\ \}| = 1$

P and Q are not the same and the magnitude of the set of factors greater than 1 of P and Q is 1.

(iii) there are infinitely many perfect squares, i.e. integers with the property that their square roots are also integers.

$\forall X \varepsilon N,\ \exists Y\ \varepsilon N\ st\ x * x = y$

For all natural numbers X, there exists a Y such that x*x = y. Since the natural numbers are infinite, the set of perfect squares will also be infinite.

2. Scheme Implementation
   Data structures: List
   Language mechanisms:
   Selection: There are multiple instances where the execution of the program changes depending on the value of an expression. In each function in the Scheme implementation there is a (cond) statement which will make a choice based on run-time values.
   Procedural Abstraction: The programmer can now use this series of functions in order to perform a merge sort operation. There are multiple helper functions that (msort) uses in order to abstract the complex collection of control.
   Recursion: The (msort), (split), and (merge) functions all have recursion.
   Exception handling and speculation: All recursive functions have base cases that prevent a stack overflow occurring.

   Haskell Implementation
   Data structures: List
   Language mechanisms:
   Selection: In there merge function the execution of the program changes depending on the value of the expression.
   Procedural Abstraction: The programmer can now use this series of functions in order to perform a merge sort operation. There is a helper function that msort uses in order to abstract the complex collection of control.
   Recursion: Both functions use recursion in order to perform the divide and conquer nature of the merge sort algorithm.
   Exception handling and speculation: All recursive functions have base cases that ensure that the function completes the calculation.

The merge sort algorithm requires the inputted list to be split multiple times. The Haskell implementation of this algorithm uses built in list operations in order to effectively do this. In the Scheme implementation, a split function is defined in order to split the list into halves. In Haskell, we can take advantage of the ‘take’ and ‘drop’ functions in order to do this. Both versions make use of helper functions and recursion in order to complete the mergesort. Both versions will also successfully return a sorted version of the passed in list.

3.

(i)

$$factors(X) = \{ I \mid \forall\ I\ <\ X - 1\ st\ X|I \}$$
$$perfect(X)\ =\ sum(factors(X))\ ==\ X$$
$$\{ X \mid \forall X\ \varepsilon\ N\ st\ perfect(X) \}$$

factors(X) defines the set of factors of X, but doesn't include X.
perfect(X) is a predicate that determines if X is a perfect number
The set includes all X in the natural numbers such that perfect(X) is true.

(ii)

```
2 factors n = [i | i <- [1..n-1], n `mod`i == 0]
1 perfect n = sum (factors n) == n
  perfects = [n | n<-[1..], perfect n]
```

This screenshot is a Haskell implementation of constructing a list of all perfect numbers. An example of the code working is below.

```
tux5: final > pwd
/home/djg365/cs360/final
tux5: final > ghci perfect.hs
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( perfect.hs, interpreted )
Ok, one module loaded.
*Main> take 3 perfects
[6,28,496]
*Main>
```

5.

```
sibling(X, Y) :- mother(M, X), mother(M, Y),
                 father(F, X), father(F, Y).
```

For all M and C, parent(M,C) ∧ female(M) -> mother(M,C)
For all F and C, parent(F,C) ∧ male(F) -> father(F,C)
For all X, Y, M and F, sibling(X,Y) -> mother(M,X) ∧ mother(M,Y) ∧ father(F,X) ∧ father(F,Y)

7.

L_digits.len := more_L_digits.len + 1

In this case, we want to add one to the current value of more_L_digits.len. This will ensure that the digit.pos value will calculate the correct value based on its position. Since the calculation for digit.val is based on 10^(digit.pos), we want L_digits.len to be greater than 0. We increase this value as we have more_L_digits so that the digit.val calculation will accurately compute the value.

L_digits.val := digit.val + more_L_digits.val

The digit calculation already takes into consideration the length of the digit. The position of an L_digit is based on its length. This allows us to simply add the values in order to get the value to carry upwards.

more_R_digits.pos := R_digits.pos - 1

R_digits is working the opposite direction of L_digits. Since the calculation for a digit is based on its position, we want to decrement an R_digits.pos value by one. This way, when the digits value is calculated, the proper value is carried upwards. Since the calculation for digit.val is based on 10^(digit.pos), we want R_digits.pos value to be negative. This will give us a decimal value for R_digits, which is the intention.

R_digits.val := digit.val + more_R_digits.val

The digit calculation already takes into consideration the position of the digit. This allows us to simply add the values in order to get the value to carry upwards.