1. 10 points: A grammar is a set of production rules for strings in a formal language. Consider the following grammar.

<stmt> ::= if ‘(‘ <expr> ‘)’ <stmt>

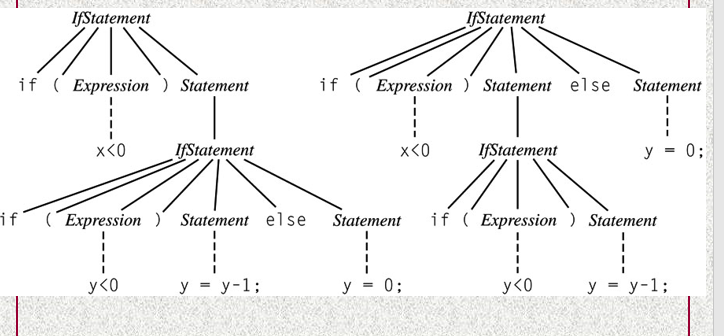
| if ‘(‘ <expr> ‘)’ <stmt> else <stmt>

1. What is the definition of an ambiguous grammar?

an ambiguous grammar is a context-free grammar for which there exists a string that can have more than one leftmost derivation or parse tree

1. Is this grammar ambiguous?

**Yes, there is an optional else statement**

1. Prove that this grammar is or is not ambiguous using parse trees.  
     
   
2. How does Haskell address the if-else ambiguity?

**Conditional expressions in Haskell must always have an else branch, which avoids the well-known dangling else problem. The dangling else problem in other languages is that it is hard to distinguish which ‘if’ statement the else belongs to.**

1. 10 points: Give the types of the following:
2. True

**Bool**

1. ‘a’

**Char**

1. 4 == 5

**Eq**

1. [1,6,3,9]

**[Int]**

1. “This test is a piece of cake!”

**String**

1. (1, ‘a’)

**(Int, Char) #of elements in a tuple is called its arity**

1. factorial n = product [1..n]

**factorial :: Integeral a => a -> a**

1. circumference r = 2 \* pi \* r

**circumference :: Float -> Float OR**

**circumference :: Double -> Double**

1. fst’ (x, \_) = x

**fst' :: (a,b) -> a**

1. max a b = if a > b then a else b

**max :: a -> a -> a**

1. 4 points: Using the :: operator, specify 20 as:
2. A bounded Integer

**20 :: Int**

1. An unbounded Integer

**20 :: Integer**

1. A floating point number

**20 :: Float**

1. A double precision floating point number.

**20 :: Double**

1. 3 points: Define a function in Haskell called divTwo that takes a number and divides it by 2.

divTwo :: (Integral a) => a -> a

**divTwo a = a / 2**

1. 3 points: Define a function in Haskell called tripleOne that takes a number, multiplies it by 3 and adds 1 to the result.

tripleOne :: (Integral a) => a -> a

**tripleOne a = (a \* 3) +1**

1. 5 points: Using an if statement, define a function in Haskell that takes a number and creates a Collatz chain. In other words, if the number is even then divide the number by two. If the number is odd then multiply it by 3 and add 1. Example: chain 10 [10,5,16,8,4,2,1]

chain :: (Integral a) => a -> [a]

**chain 1 = [1]**

**chain a = if (a `mod` 2 == 0)**

**then a:chain ( a `div` 2 )**

**else a:chain ((a \* 3) +1)**

1. 5 points: Rewrite chain as above using guards instead of an if statement.

Chain’ :: (Integral a) => a -> [a]

**Chain’ 1 = [1]**

**Chain’ a | (a `mod` 2 == 0) = a:chain’ ( a `div` 2 )**

**| otherwise = a:chain’ ((a \* 3) +1)**

1. 5 points: Write a command to find the Collatz chains for all numbers between 1 and 100. Use your chain function.

**map (chain') [1..100]**

1. 5 points: Use list comprehensions to convert all uppercase letters to lowercase letters. You may use toLower from Data.Char **IF YOU IMPORT IT FIRST**.

**Import Data.Char (toLower)**

**[toLower a | a <- [‘A’..’Z’]**

1. 10 points: Using recursion, define the replicate function.

replicate’ :: (Num i, Ord i) => i -> a -> [a]

**replicate' 0 y = []**

**replicate' x y = y : replicate' (x-1) y**

1. 5 points: What is the definition of a curried function?

**All functions in Haskell are assumed to be curried, which means they can return a function as the result. Mult x y z means ((mult x) y) z.**

1. 5 points: Give an example of a section (partially applied function). Use + or \*.

**add' :: Int -> Int -> Int**

**add' x y = x + y**

**addOne = add' 1**

1. 5 points: Rewrite divTwo using a partially applied function.

divTwo :: (Integral a) => a -> a

divTwo a = a div 2

divTwo’ = divTwo

1. 5 points: Write a function to find the factors of a number using list comprehension

factors :: (Integral a) => a -> [a]

**factors a = [b | b <-[1..a], a `mod` b == 0]**

1. 5 points: Write a function to check and see if a number is prime. You may use the factors function defined above.

**isPrime :: (Integral a) => a -> Bool**

**isPrime a = (factors a) == [1,a]**

1. 5 points: Write a command to find the largest prime number less than 1,000,000.

**[a | a <- [1..100], isPrime a]**

1. 5 points: Define a function called factorial using folds.

factorial :: (Num a, Enum a) => a -> a

**factorial' a = foldl (\*) 1 [1..a]**

1. 5 points: What is the difference between type and data in Haskell

*:* A type is a prexisiting data type in Haskell. Examples would include: Bool, Int, Float, etc. A data is a completely new type, for example, if Bool did not already exist in Haskell, data Bool = False | True would be the data declaration for Bool.

1. 10 points: You have an array of 5 different data models that take in a number n and return a prediction. Using function application ($) and map, write a function to get predictions from your five models.

let models = [model1, model 2, model3, model4, model5]

getPredictions :: a -> [a -> b] -> [b]

**getPredictions n = map ($ n) [models]**

1. 10 points: Define a new data type for the days of the week. You should be able to compare equality, compare order (i.e. Monday < Tuesday), convert them to and from strings (for printing etc.) and create lists of days.

data Day = **Monday | Tuesday | Wednesday | Thursday |Friday | Saturday | Sunday**

**deriving (Eq, Ord, Show, Read, Bounded, Enum)**