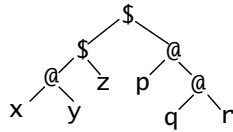


1. [10] Given the following abstract syntax tree



Translate it to infix, using no parentheses, and specify each operator's associativity and its binding power (i.e., its precedence) as it would be specified in a Haskell script.

```
x @ y $ z $ p @ q @ r
infixl 0 $
infixr 1 @
```

2. [15] Rewrite the following Haskell function definition

```
(!!) :: [b] -> Int -> b
(x:_) !! 0 = x
(_:xs) !! n | n>0 = xs !! (n-1)
(_:_) !! _ = error "(!!): negative index"
_ !! _ = error "(!!): index too large"
```

without pattern-matching in two different ways, preserving the tests' order:

- a. [10] using guards

```
xs !! n
| not (null xs) && n==0 = head xs
| not (null xs) && n > 0 = (tail xs) !! (n-1)
| not (null xs) = error "(!!): negative index"
| otherwise = error "(!!): index too large"
```

- b. [5] using conditional expressions

```
xs !! n = if not (null xs) && n==0 then head xs
          else if not (null xs) && n > 0 then (tail xs) !! (n-1)
          else if not (null xs) then error "(!!): negative index"
          else error "(!!): index too large"
```

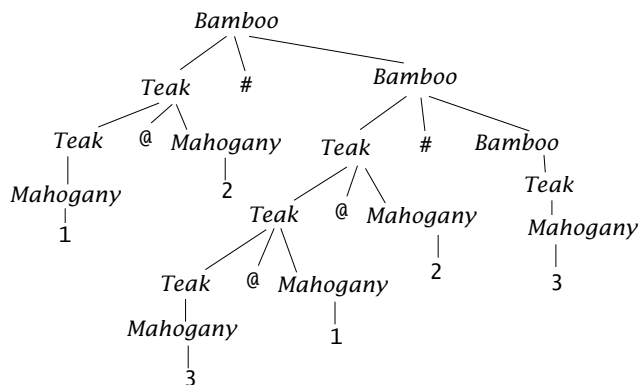
3. [20] Given the expression

1 @ 2 # 3 @ 1 @ 2 # 3

and the grammar

```
Bamboo ::= Teak # Bamboo | Teak
Teak ::= Mahogany | Teak @ Mahogany
Mahogany ::= 1 | 2 | 3 | 4 | 5 | 6
```

- a. [10] draw a parse tree for the expression



- b. [5] translate the expression to postfix

```
1 @ 2 # 3 @ 1 @ 2 # 3
((1 @ 2) # (((3 @ 1) @ 2) # 3))
((1 2 @) (((3 1 @) 2 @) 3 #) #)
1 2 @ 3 1 @ 2 @ 3 # #
```

- c. [5] show the steps in the evaluation of the postfix expression using a stack, given these definitions of the operators (#) and (@):

$$x @ y = x + 2^*y$$

$$x \# y = 2^*x + y$$

```

      1 2 @ 3 1 @ 2 @ 3 # #
1     2 @ 3 1 @ 2 @ 3 # #
1 2   @ 3 1 @ 2 @ 3 # #
5     3 1 @ 2 @ 3 # #
5 3   1 @ 2 @ 3 # #
5 3 1 @ 2 @ 3 # #
5 5   2 @ 3 # #
5 5 2 @ 3 # #
5 9   3 # #
5 9 3 # #
5 2 1 #
3 1

```

4. [15] Show the steps in the evaluation of the expression

$f(3 > 0) (3 + 4 + 5) (6 + 7 + 8)$

where f is defined by

$f x y z = \text{if } x \text{ then } y^*y \text{ else } z^*z$

- a. Use leftmost innermost evaluation.

```

f (3>0) (3+4+5) (6+7+8)
~> f True (3+4+5) (6+7+8)
~> f True (7+5) (6+7+8)
~> f True 12 (6+7+8)
~> f True 12 (13+8)
~> f True 12 21
~> if True then 12*12 else 21*21
~> 12*12
~> 144

```

- b. Use leftmost outermost (non-lazy) evaluation.

```

f (3>0) (3+4+5) (6+7+8)
~> if (3>0) then (3+4+5)*(3+4+5) else (6+7+8)*(6+7+8)
~> if True then (3+4+5)*(3+4+5) else (6+7+8)*(6+7+8)
~> (3+4+5)*(3+4+5)
~> (7+5)*(3+4+5)
~> 12*(3+4+5)
~> 12*(7+5)
~> 12*12
~> 144

```

- c. Use leftmost lazy evaluation.

```

f (3>0) (3+4+5) (6+7+8)
~> let x = 3>0, y = 3+4+5, z = 6+7+8 in if x then y*y else z*z
~> let x = True, y = 3+4+5, z = 6+7+8 in if x then y*y else z*z
~> let x = True, y = 3+4+5, z = 6+7+8 in if True then y*y else z*z
~> let y = 3+4+5 in y*y
~> let y = 7+5 in y*y
~> let y = 12 in y*y
~> 12*12
~> 144

```

5. [10] If sets are represented by ordered lists, then `merge` implements *set union*. Define a Haskell function that implements *set intersection* for sets of `Int`. For example,

```
intersection [0..5] [3..8] ==> [3,4,5]
```

A solution:

```
> intersection :: Ord a => [a] -> [a] -> [a]
> intersection [] _ = []
> intersection _ [] = []
> intersection (x:xs) (y:ys)
> | x == y = x : intersection xs ys
> | x < y = intersection xs (y:ys)
> | y < x = intersection (x:xs) ys
```

6. [15] Define a function `showMoney` so that if `cents` is an `Integer` denoting an amount of money in cents, `showMoney cents` is a `String` displaying the amount in the conventional format. The result's first digit should be preceded by a dollar sign, and the result should contain a decimal point preceded by at least one digit and followed by exactly two digits. A negative amount should be enclosed in parentheses, and should not have a minus sign. Some examples:

```
showMoney 12345 ==> "$123.45"
showMoney 23    ==> "$0.23"
showMoney 0     ==> "$0.00"
showMoney (-678) ==> "($6.78)"
```

Your definition should use local definitions and/or auxiliary functions for clarity and simplicity.

A solution:

```
> showMoney :: Integer -> String
> showMoney c
> | c < 0 = "(" ++ do1Cts ++ ")"
> | otherwise = do1Cts
> where
> s = show (abs c)
> digits = case s of
>     [] -> "00" ++ s
>     [_,_] -> "0" ++ s
>     _ -> s
> do1Cts = "$" ++ ds ++ "." ++ cs
> (ds,cs) = splitAt (length digits - 2) digits
```

A nice alternative solution:

```
> showMoney :: Integer -> String
> showMoney c
> | c < 0 = "(" ++ showMoney (abs c) ++ ")"
> | c < 10 = "$0.0" ++ show c
> | c < 100 = "$0." ++ show c
> | otherwise = "$" ++ show (c `div` 100) ++ show2d (c `mod` 100)
> where
> show2d n = (if n < 10 then ".0" else ".") ++ show n
```