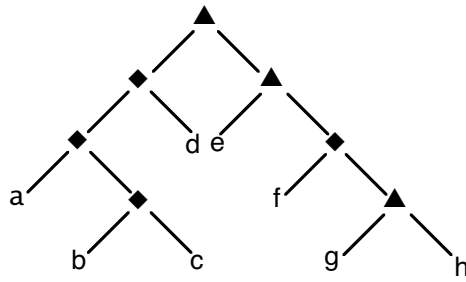


1. [12] Given this abstract syntax tree



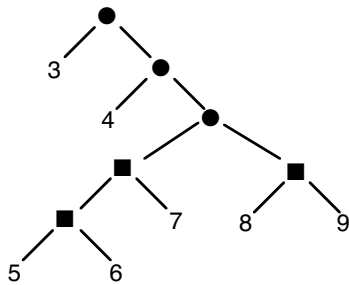
write the expression it represents in each of three notations

- a. prefix    ▲ ◆ ◆ a ◆ b c d ▲ e ◆ f ▲ g h
- b. postfix   a b c ◆ ◆ d ◆ e f g h ▲ ◆ ▲ ▲
- c. infix     ((a ◆ (b ◆ c)) ◆ d) ▲ (e ▲ (f ◆ (g ▲ h)))

2. [12] Given this expression

3 ● 4 ● 5 ■ 6 ■ 7 ● 8 ■ 9

and this abstract syntax tree



indicate the operators' precedence and associativity by circling the appropriate entries in this table:

●	left	right	high	low
■	left	right	high	low

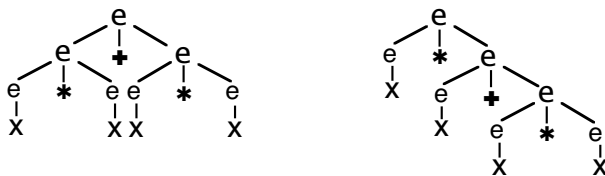
3. [12] Given the following BNF grammar,

$\langle e \rangle ::= \langle e \rangle + \langle e \rangle \mid \langle e \rangle * \langle e \rangle \mid x$

draw a parse tree for this expression

$x * x + x * x$

with respect to the grammar. If the grammar is ambiguous, prove it by drawing another parse tree.



4. [12] Given the definition  $f\ x = x + x$ , evaluate the expression  $f( f(3*5) )$  using

a. innermost evaluation.

```
f( f(3*5) )
⇒ f( f(15) )
⇒ f( 15+15 )
⇒ f( 30 )
⇒ 30+30
⇒ 60
```

b. outermost evaluation.

```
f( f(3*5) )
⇒ f(3*5) + f(3*5)
⇒ (3*5 + 3*5) + f(3*5)
⇒ (15 + 3*5) + f(3*5)
⇒ (15 + 15) + f(3*5)
⇒ 30 + f(3*5)
⇒ 30 + (3*5 + 3*5)
⇒ 30 + (15 + 3*5)
⇒ 30 + (15 + 15)
⇒ 30 + 30
⇒ 60
```

c. lazy evaluation.

```
f( f(3*5) )
⇒ let x = f(3*5) in x + x
⇒ let x = (let x = 3*5 in x + x) in x + x
⇒ let x = (let x = 15 in x + x) in x + x
⇒ let x = 15 + 15 in x + x
⇒ let x = 30 in x + x
⇒ 30 + 30
⇒ 60
```

5. [15] Define a Haskell function `day` so that if  $n :: \text{Int}$  and  $1 \leq n \leq 7$ , then `day n` is the name of the  $n$ th day of the week. For example, `day 3 = "Tue"`. If  $n < 1$  or  $7 < n$ , your function should halt with a sensible error message.

a. Define the function using a **case** expression.

```
> day :: Int -> String
> day n = case n of
>     1 -> "Sun"
>     2 -> "Mon"
>     3 -> "Tue"
>     4 -> "Wed"
>     5 -> "Thu"
>     6 -> "Fri"
>     7 -> "Sat"
>     _ -> error ("day: argument (" ++ show n ++ ") out of range")
```

b. Define the function using literal parameters.

```
> day 1 = "Sun"
> day 2 = "Mon"
> day 3 = "Tue"
> day 4 = "Wed"
> day 5 = "Thu"
> day 6 = "Fri"
> day 7 = "Sat"
> day n = error ("day: argument (" ++ show n ++ ") out of range")
```

c. Define the function using a list of day names and the operator `(!!)`.

```
> day n
> | 1 <= n && n <= 7 = ["Sun","Mon","Tue","Wed","Thu","Fri","Sat"] !! (n-1)
> | otherwise       = error ("day: argument (" ++ show n ++ ") out of range")
```

6. [12] Define in Haskell a function `uniques :: [Int] -> [Int]` so that `uniques xs` is a list containing only those elements of `xs` that occur in `xs` exactly once. Do not use explicit recursion— use list comprehensions.

For example,

```
uniques [1,2,3,4,3,2,1] = [4]
uniques [1,2,3,3,2,1]   = []
uniques [1,2,3,4,5,6]  = [1,2,3,4,5,6]
```

A solution:

```
> uniques xs = [ x | x <- xs, length [ y | y <- xs, y == x ] == 1 ]
```

7. [15] This question concerns a function `cpf` which produces a `String`, representing a dollar amount, in a form suitable for printing on a check. Define `cpf` so that `cpf w c` is a `String` of length `w` (or more, if necessary); its first character, '\$', should be followed by enough '\*' characters so that the amount is right-justified in the `String`. Finally, the amount `c` is a positive `Integer` representing *cents*, but `cpf` should display it in the usual decimal-dollar format, with a decimal point followed by two digits.

For example,

```
cpf 10 250 = "$*****2.50"
cpf 10 2   = "$*****0.02"
cpf 1 250  = "$2.50"
```

This problem is not inherently difficult. Think before you start coding, and make good use of local definitions.

A solution:

```
> cpf :: Int -> Integer -> String
> cpf w c = "$" ++ stars ++ amt
>   where
>     amt = dollars ++ "." ++ cents
>     dollars = show (c `div` 100)
>     cents  = (if cts < 10 then "0" else "") ++ show cts
>     cts = c `mod` 100
>     stars = replicate (w - 1 - length amt) '*'
```