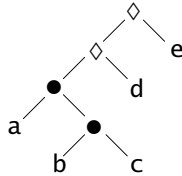


1. [10] Consider this infix expression:

$a \bullet b \bullet c \diamond d \diamond e$

and this abstract syntax tree for the same expression:



a. Express the operators' associativity and precedence as they could be expressed in a Haskell script.

```
infixr 2 •
infixl 1 ◊
```

b. Define a BNF grammar for expressions with these operators. Use the names *diamond* and *dot* for the nonterminals whose productions contain the corresponding operators, and assume that you are given the productions

$$\langle \text{letter} \rangle ::= a \mid b \mid \dots \mid z$$

The additional productions:

$$\begin{aligned} \langle \text{diamond} \rangle & ::= \langle \text{diamond} \rangle \diamond \langle \text{dot} \rangle \mid \langle \text{dot} \rangle \\ \langle \text{dot} \rangle & ::= \langle \text{letter} \rangle \bullet \langle \text{dot} \rangle \mid \langle \text{letter} \rangle \end{aligned}$$

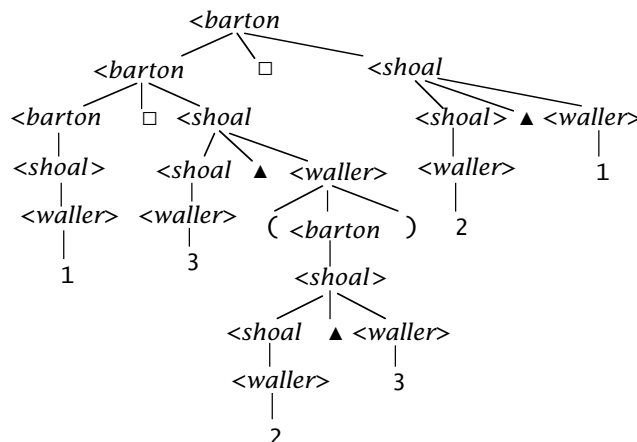
2. [15] Given the grammar

$$\begin{aligned} \langle \text{barton} \rangle & ::= \langle \text{shoal} \rangle \mid \langle \text{barton} \rangle \square \langle \text{shoal} \rangle \\ \langle \text{shoal} \rangle & ::= \langle \text{waller} \rangle \mid \langle \text{shoal} \rangle \blacktriangle \langle \text{waller} \rangle \\ \langle \text{waller} \rangle & ::= 1 \mid 2 \mid 3 \mid (\langle \text{barton} \rangle) \end{aligned}$$

draw a parse tree for the expression

$1 \square 3 \blacktriangle (2 \blacktriangle 3) \square 2 \blacktriangle 1$

The solution:



3. [15] Show the steps in the evaluation of the expression

$f(3+5)(9*4)(12-6)$

where f is defined by

$f\ x\ y\ z = \text{if } y < 0 \text{ then } y * x \text{ else } y * z$

assuming that the default evaluation strategy is

- a. lazy

$f(3+5)(9*4)(12-6)$

~ $\text{let } x = 3+5, y = 9*4, z = 12-6 \text{ in if } y < 0 \text{ then } y * x \text{ else } y * z$

~ $\text{let } x = 3+5, y = 36, z = 12-6 \text{ in if } y < 0 \text{ then } y * x \text{ else } y * z$

~ $\text{let } x = 3+5, y = 36, z = 12-6 \text{ in if } 36 < 0 \text{ then } y * x \text{ else } y * z$

~ $\text{let } x = 3+5, y = 36, z = 12-6 \text{ in if False then } y * x \text{ else } y * z$

~ $\text{let } y = 36, z = 12-6 \text{ in } y * z$

~ $\text{let } y = 36, z = 6 \text{ in } y * z$

~ $36 * 6$

~ 216

- b. innermost

$f(3+5)(9*4)(12-6)$

~ $f\ 8\ (9*4)\ (12-6)$

~ $f\ 8\ 36\ (12-6)$

~ $f\ 8\ 36\ 6$

~ $\text{if } 36 < 0 \text{ then } 36 * 8 \text{ else } 36 * 6$

~ $\text{if False then } 36 * 8 \text{ else } 36 * 6$

~ $36 * 6$

~ 216

- c. outermost

$f(3+5)(9*4)(12-6)$

~ $\text{if } (9*4) < 0 \text{ then } (9*4) * (3+5) \text{ else } (9*4) * (12-6)$

~ $\text{if } 36 < 0 \text{ then } (9*4) * (3+5) \text{ else } (9*4) * (12-6)$

~ $\text{if False then } (9*4) * (3+5) \text{ else } (9*4) * (12-6)$

~ $(9*4) * (12-6)$

~ $36 * (12-6)$

~ $36 * 6$

~ 216

4. [10] Rewrite the following function definitions using no guards, no patterns, and only conditional (**if-then-else**) expressions.

> $f4\ 3\ _ \ 5 = 8$

> $f4\ n\ []\ y = 12*y$

> $f4\ _ [a]\ _ = 77$

> $f4\ 6\ _ \ y$

> | $y < 0 = 9*y$

> | $y > 0 = 15$

A solution:

> $f4\ n\ \text{as } y = \text{if } n == 3 \ \&\& \ y == 5 \ \text{then } 8$

> | $\text{else if null as then } 12*y$

> | $\text{else if null (tail as) then } 77$

> | $\text{else if } n == 6 \ \text{then if } y < 0 \ \text{then } 9*y$

> | | $\text{else if } y > 0 \ \text{then } 15$

> | | $\text{else error "f4: no true guard"}$

> | $\text{else error "f4: no pattern matches"}$

5. [10] Is the following function definition tail-recursive? Circle one: **yes** **no**

```
> power :: Int -> Int -> Int
> power x n = if n==0 then 1 else x * power x (n-1)
```

If you circled *yes*, give a non-tail-recursive definition of the same function.

If you circled *no*, give a tail-recursive definition of the same function.

```
> power1 :: Int -> Int -> Int
> power1 x n = pow n 1
>   where
>     pow n p = if n==0 then p else pow (n-1) (p*x)
```

6. [15] Suppose that sets of `Int` are represented in a Haskell program by strictly ascending lists of type `[Int]`. Define a function `subset :: [Int] -> [Int] -> Bool` so that `subset ps qs` returns `True` if the set represented by `ps` is a subset of the set represented by `qs`, and `False` otherwise. For full credit, your solution's worst-case running time must be linear in the lengths of its arguments.

```
> subset :: [Int] -> [Int] -> Bool
> subset [] _ = True
> subset (_:_) [] = False
> subset (x:xs) (y:ys)
>   | x < y = False
>   | x > y = subset (x:xs) ys
>   | otherwise = subset xs ys
```

7. [10] Define a function `sums` so that given two lists `cs` and `ds`, both of type `[Int]`, `sums cs ds` is a list of sums of corresponding elements. For example,

```
sums [1,4,3] [4,2,4,7] ~ [5,6,7]
```

Your definition should **not** use recursion explicitly.

A solution:

```
> sums :: [Int] -> [Int] -> [Int]
> sums cs ds = [ c+d | (c,d) <- zip cs ds ]
```

Generalize your definition of `sums` to define a function `zipWith`, so that `zipWith f xs ys`, where `xs` and `ys` are lists, returns a list of applications of `f` to corresponding elements of `xs` and `ys`. For example,

```
zipWith (-) [1,4,3, 8] [4,2,2] ~ [-3,2,1]
```

Include `zipWith`'s most general type.

```
> zipWith :: (a->b->c) -> [a] -> [b] -> [c]
> zipWith f xs ys = [ f x y | (x,y) <- zip xs ys ]
```

Then redefine `sums` using `zipWith`.

```
> sums = zipWith (+)
```