Haskell for Readers

Joachim Breitner[*], DFINITY Foundation[†]

Solution 1

Both operators are left-associative, so that `10 - 2 + 3 - 4` means `((10 - 2) + 3) - 4` as expected.

Solution 2

The precedences of `(+)` and `(-)` are the same, and smaller than the precedence of `(*)`, which is again smaller than the precedence of `(^)`.

Solution 3

`7`

Solution 4

`3`

Solution 6

```
absoluteValue x = if x < 0 then - x else x
```

Solution 7

```
isHalfRound x = x `mod` 10 == 0 || x `mod` 10 == 5
```

Solution 8

```
isEven x = x `mod` 10 == 0 || x `mod` 10 == 2 || x `mod` 10 == 4 || x `mod` 10 == 6 || x `mod` 10 == 8
```

Solution 9

```
sumDigits n = if n < 10 then n else sumDigits (n `div` 10) + (n `mod` 10)
```

Solution 10

```
fixEq f x = if x == f x then x else fixEq f (f x)
```

Solution 11

```
isMultipleOf3 x = fixEq sumDigits x == 3 || fixEq sumDigits x == 6 || fixEq sumDigits x == 9
```

Solution 12

The definitions for `countCountDigits` and `sumSumDigits`:

```
Prelude> countCountDigits = twice countCountDigits
Prelude> sumSumDigits = twice sumDigits
```

[*] http://www.joachim-breitner.de/
[†] https://dfinity.org/

Solution 13

```
id :: a -> a
```

Solution 14

It is the function `flip` that takes a function and swaps its first two arguments.

Solution 15

This type is legal. Every value of type `Wat` is built from the constructor `Wat`, applied to another value of type `Wat`. So, unless there are exceptions or nontermination around, it is just an infinite tower of `Wat`s:

```
wat :: Wat
wat = Wat wat
```

Solution 16

There are four: `Nothing`, `Just Nothing`, `Just (Just False)` and `Just (Just True)`. It can be useful if, for example, the outer `Maybe` indicates whether some input was *valid*, whereas `Just Nothing` could indicate that the input was valid, but empty. But arguably this is not best practice, and dedicated data types with more speaking names could be preferred here.

Solution 17

```
Maybe (Integer, Integer)
```

Solution 18

```
fromEitherUnit :: Either () a -> Maybe a
fromEitherUnit (Left ()) = Nothing
fromEitherUnit (Right x) = Just x

toEitherUnit :: Maybe a -> Either () a
toEitherUnit Nothing = Left ()
toEitherUnit (Just x) = Right x
```

We can make `fromEitherUnit` more polymorphic; it can simply ignore the argument to `Left`. We cannot do this in `toEitherUnit`: we would not have a value of type `b` at hand to pass to `Left`.

Solution 19

```
Pondering the question...
42
```

The line `return 23` doesn't do anything: There is no side-effect, and the result (the value `23`) is not bound to any variable and hence ignored.

Solution 20

```
Hooray!
Hooray!
Hooray!
Almost done
Hooray!
Done
Done
```

Note that the `putStrLn "And up she rises."` is never executed.

Solution 21

```
instance Eq Employee where
    e1 == e2 =
        name e1 == name e2 &&
        room e1 == room e2 &&
        pubkey e1 == pubkey e2
```

The problem is: If the record gains additional fields, this code still compiles, and the programmer is not warned that they should update it. By not using the record accessors, and using normal constructor syntax instead, this can be avoided:

```
instance Eq Employee where
    Employee n1 r1 p1 == Employee n2 r2 p2 =
        n1 == n2 && r1 == r2 && p1 == p2
```

Solution 22

The `Semigroup` type class is defined as

```
class Semigroup a where
  (<>) :: a -> a -> a
```

with the additional requirement that the `(<>)` operation is associative.

Solution 23

There are (at least) two sensible instances for the `Semigroup` type class for trees:

1. The first one concatenates two trees, so that an in-order traversal first visits the value of the left and then of the right tree:

   ```
   instance Semigroup (Tree a) where
       Leaf <> t = t
       (Node x l r) t = Node x l (r <> t)
   ```

   There are variations of this code that are more likely to produce a balance tree – although then it might be that associativity holds when one considers different shapes of the same data equivalent (which is commonly the case for search trees).

2. Another one traverses both trees in parallel, using a `Semigroup` instance for the elements to combine values that are present in boht trees:

   ```
   instance Semigroup a => Semigroup (Tree a) where
       Leaf <> t = t
       t <> Leaf = t
       Node x l1 r1 <> Node y l2 r2 = Node (x <> y) (l1 <> l2) (r1 <> r2)
   ```

   Here, the instance head itself has a constraint: This instance for `Tree a` only exists if there is a `Semigroup` instance for the type of values.

Which instance is the right one? That depends on the purpose of the `Tree` data structure in the code; and maybe neither is the right one, in which case it might be better to have *no* instance at all, and use normal functions for these operations.

Solution 24

One might expect this code:

```
summarize :: Monoid a => Tree a -> a
summarize Leaf = mempty
summarize (Node x l r) = summarize l <> x <> summarize r
```

(at least given everything we did so far considered in-order traversals.)

If we have two trees with the same elements in the same order, but in a different shape, e.g.

```
t1 = Tree x Leaf (Tree y Leaf (Tree z Leaf))
t2 = Tree y (Tree x Leaf Leaf) (Tree z Leaf Leaf)
```

for some values x, y and z, then we will always have `summarize t1 = summarize t2`, assuming that the `Monoid` instance for the type of values is lawful. This is important if we use these trees as search trees, where the internal shape should be an implementation detail that should not be visible from the outside.

Solution 27

```
forM :: Monad m => [a] -> (a -> m b) -> m [b]
forM [] f = return []
forM (x:xs) f = do
    y <- f x
    ys <- forM xs f
    return (y : ys)
```

Solution 25

```
import Prelude hiding (fmap, (<$>))
(>>) :: Monad m => m a -> m b -> m b
a >> b = a >>= (\_ -> b)
fmap :: Monad f => (a -> b) -> f a -> f b
fmap f a = a >>= (return . f)
(<$>) :: Monad f => (a -> b) -> f a -> f b
(<$>) = fmap
(<$) :: Monad f => a -> f b -> f a
x <$ a = const x <$> a
liftA2 :: Monad f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = a >>= (\x -> b >>= (\y -> return (f x y)))
(<*>) :: Monad f => f (a -> b) -> f a -> f b
(<*>) = liftA2 (\f x -> f x)
(<*) :: Monad f => f a -> f b -> f a
(<*) = liftA2 (\x y -> x)
(*>) :: Monad f => f a -> f b -> f b
(*>) = liftA2 (\x y -> y)
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
(a >=> b) x = a x >>= b
join :: Monad m => m (m a) -> m a
join a = a >>= id
```

Solution 26

Since `when False action` is not supposed to execute `action`, it has no way of producing a `m a`. But it can always create a `m ()` using `return ()`.

Solution 28

The instance is

```
instance Monoid r => Applicative ((,) r)
```

and the `Monoid` constraint on `r` gives us exactly the operations needed to to implement `pure` and `(<*>)`. Try it yourself!