Declarative Languages Haskell: ASCII Boxes

Some practicalities

- In the folder `/localhost/examen/loginnaam/OPGAVEN`, you'll find the files `AsciiBoxes.hs` and `AsciiBoxesTest.hs`.
  - The file `AsciiBoxes.hs` contains a template for your solution.
  - First of all, fill in your name, student number and major.

    ```
    -- Jan Jansen
    -- r0123456
    -- master cs
    ```

  - At the top of the file, a number of functions that *might* come in handy have already been imported. **Look up how they work** if you're not familiar with them. You can also **import extra functions and types**.
  - For each assignment, a number of functions and type classes have already been defined with corresponding type signatures. These functions and type signatures **may not be modified**. Replace all occurrences of `undefined` with your implementation. You can add arguments in front of the equals sign, but, when possible, try to write the function *point-free*. It is of course also permitted to add extra helper functions.
  - Even though you will develop a whole program in this exam, you can make most assignments separately. **Whenever you're stuck on an assignment, try the next**.
  - You can test your solution using `AsciiBoxesTest.hs`. Run the following command in the directory you put the two `.hs`-files in:

    ```
    runhaskell AsciiBoxesTest.hs
    ```

    N.B. the fact that all your tests pass doesn't mean that your program is completely correct, nor that you will get the maximum score.

- To hand in your final solution, copy the file
  `/localhost/examen/loginnaam/OPGAVEN/AsciiBoxes.hs`
  to the file
  texttt/localhost/examen/loginnaam/haskell.hs.

ASCII Boxes

For this exam we will use an infinite grid like the one shown in Figure 1. The origin of the grid is cell (point) $(0, 0)$. In general, every cell in the grid can be represented in Haskell with a tuple of two `Int`s (i.e., we assume integral coordinates only): The first integer represents the position on the X-axis and the second integer the position on the Y-axis:

```
type Point  = (Int, Int) -- X, Y
```

For example, the origin `(0,0)` is shown with gray and the point `(4,1)` with black, in Figure 1.

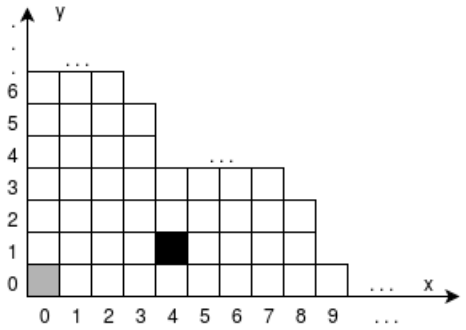


Figure 1: The grid

The goal of this exam is to write a small library for defining, manipulating and printing rectangles (which we call *ASCII Boxes*) on the plane. We can define a box as follows:

```
type Width  = Int
type Height = Int
data Box = Box (Width, Height) (Point -> Char)
```

A box is represented by:

- Its width,
- Its height,
- A rendering function of type `(Point -> Char)`, which expresses how each point inside the rectangle is printed.

Note that we do not store a rectangle's position, because we assume that its leftmost downmost point is the origin `(0,0)` of the grid. For example, `(Box (8,3) (\(x,y) -> 'a'))` represents the following box:

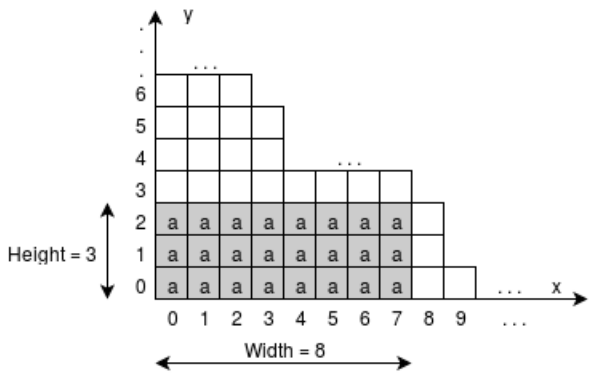


Figure 2: An ASCII Box

Part 1: Printing ASCII Boxes

Task 1a. Define a rendering function for ASCII Boxes, that is, a function that transforms a box to a string:

```
renderBox :: Box -> String
```

For example:

```
ghci> renderBox (Box (8,3) (\(x,y) -> 'a'))
"aaaaaaaa\naaaaaaaa\naaaaaaaa\n"
```

Every row of the box should be terminated by a newline, so that the box can be nicely printed in a terminal. For example, printing the above box gives:

```
ghci> putStrLn (renderBox (Box (8,3) (\(x,y) -> 'a')))
aaaaaaaa
aaaaaaaa
aaaaaaaa
```

Task 1b. Give an instance for class `Show` for datatype `Box`.

```
class Show a where
  show :: a -> String
```

For example:

```
ghci> show (Box (3,2) (\(x,y) -> 'b'))
"bbb\nbbb\n"
```

Part 2: Basic Boxes

Task 2a. Define the empty box, which has no rows and no columns:

```
emptyBox :: Box
```

For example:

```
ghci> renderBox emptyBox
""
```

Note: As you can see in the example, the `emptyBox` is rendered as `""` and not as `"\n"`. You should add a newline at the end of every row of a box but the empty box has zero rows!

Task 2b. Define the function `constantBox`:

```
constantBox :: (Width, Height) -> Char -> Box
```

The `constantBox` function takes the width `w`, the height `h`, and a character `c` and constructs a box of the given dimensions where all points inside it appear as character `c`. For example, `(constantBox (9,3) 'g')` represents the following box:

```
ghci> constantBox (9,3) 'g'
ggggggggg
ggggggggg
ggggggggg
```

Part 3: Reasoning about Points

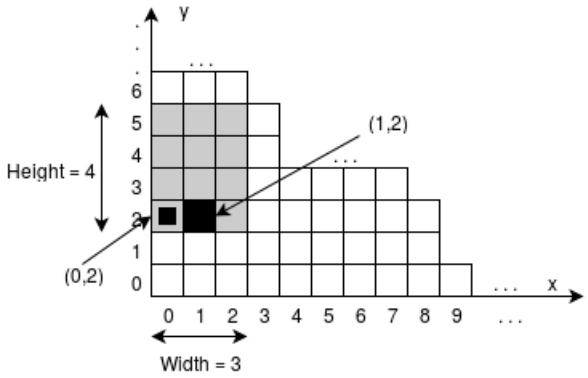
Define the function `inArea`:

```
inArea :: Point -> Point -> (Width, Height) -> Bool
```

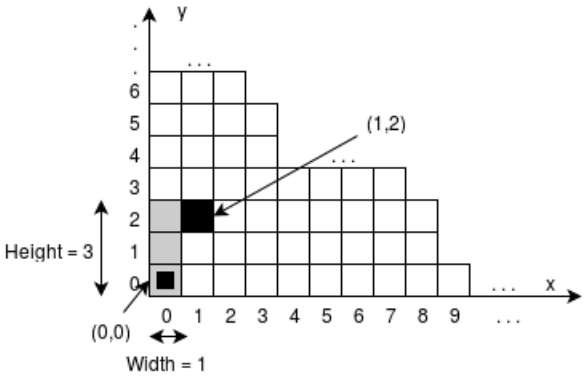
that checks whether a point is *inside* a given area. A call `(inArea p ldp (w,h))` returns `True` iff point `p` is inside the rectangle with width `w` and height `h`, where the leftmost-downmost corner is `ldp`. For example:

```
ghci> inArea (1,2) (0,2) (3,4)
True
ghci> inArea (1,2) (0,0) (1,3)
False
```

Figure 3 shows the above examples graphically:



`inArea (1,2) (0,2) (3,4) == True`



`inArea (1,2) (0,0) (1,3) == False`

Figure 3: `inArea` examples

Part 4: ASCII Box Combinators

Task 4a. Define the combinator `beside` that puts two boxes next to each other to create a bigger box:

```
beside :: Box -> Box -> Box
```

If the two boxes do not have the same height, the height of the resulting box will be the maximum of the input heights. Note that the space should be added **above** the box with the smaller height. For example:

```
ghci> beside (constantBox (1,1) 'b') (constantBox (3,2) 'a')
␣aaa
baaa
```

Note: By space we mean that the rendering function should return the space character (' ') in the area not filled by the argument boxes. The spaces are explicitly shown in the above example with character (␣).

Task 4b. Define the combinator `above` that takes two boxes and puts the first above the second to create a bigger box:

```
above :: Box -> Box -> Box
```

If the two boxes do not have the same width, the width of the resulting box will be the maximum of the input widths. Note that the space should be added **to the right** of the box with the smaller width. For example:

```
ghci> above (constantBox (1,1) 'b') (constantBox (4,2) 'a')
b␣␣␣
aaaa
aaaa
```

Note: By space we mean that the rendering function should return the space character (' ') in the area not filled by the argument boxes. The spaces are explicitly shown in the above example with character (␣).

Task 4c. Define the combinator `wrapBox` that takes a character and a box and wraps the box with the given character:

```
wrapBox :: Char -> Box -> Box
```

For example:

```
ghci> wrapBox '*' (constantBox (3,2) 'a')
*****
*aaa*
*aaa*
*****
```

Hint: You can implement `wrapBox` as a combination of functions `beside` and `above`.

Task 4d. Define the combinator `overlay` that takes two boxes and overlays the second with the first (puts the first box on top of the second):

```
overlay :: Box -> Box -> Box
```

The resulting box should have width equal to the maximum width of the argument boxes and height equal to the maximum height of the argument boxes. The area not filled by any of the argument boxes should be rendered as the space character `(' ')`. For example:

```
ghci> overlay (constantBox (3,4) 'a') (constantBox (5,2) 'b')
aaa␣␣
aaa␣␣
aaabb
aaabb
```

Note: For clarity, the spaces are explicitly shown in the above example with character (␣).

Graphically, the same example is also presented below:

```
                aaa                 aaa␣␣
overlay         aaa                 aaa␣␣
                aaa   bbbbb    =    aaabb
                aaa   bbbbb         aaabb
```

Task 4e. Define the following combinator that takes a list of boxes and creates a box by putting the boxes next to each other (in the order they are given).

```
besideMany :: [Box] -> Box
```

For example:

```
ghci> besideMany [constantBox (1,3) 'a', constantBox (2,1) 'b', constantBox (2,4) 'c']
␣␣␣cc
a␣␣cc
a␣␣cc
abbcc
```

Additionally, `(besideMany [])` should yield the `emptyBox`. If the given list has a single element then the result should be just that element.
Hint: You can use the function `beside` from Task 4a.

Part 5: A Class for Boxes

We define a class for values that can be represented by a `Box` as follows:

```
class Boxable a where
  toBox :: a -> Box
```

Every type that can be converted to a `Box` can be made an instance of the `Boxable` class.

Task 5a. Give an instance for class `Boxable` for type `Char`:

```
instance Boxable Char where
```

The instance should create a box containing just the given character:

```
ghci> renderBox (toBox 'a')
"a\n"
```

Task 5b. Given a type `a` that is an instance of `Boxable`, give an instance for `[a]`:

```
instance Boxable a => Boxable [a] where
```

The instance should simply convert every element of the list to a box and put them beside each other to form a single box. For example (a `String` is a list of `Char`s):

```
ghci> renderBox (toBox "Hello World!!")
"Hello World!!\n"
```

Part 6: Making Histograms

Define function `makeHistogram`:

```
makeHistogram :: [(Char, Int)] -> Box
```

Function `makeHistogram` takes a list of tuples where each tuple contains:

- A character, to be used as a label for the column of the histogram and
- An integer, which is the height of the column.

Every column has a width of 3 cells and it consists of the character `#`. Each label is enclosed in parenthesis (so the total width of the label is also 3) and there should be a column of spaces of width 1 between the columns. For example, the call `(makeHistogram [('a', 5), ('b', 10), ('c', 7)])` should have the following result:

```
ghci> makeHistogram [('a', 5), ('b', 10), ('c', 7)]
␣␣␣␣###␣␣␣␣
␣␣␣␣###␣␣␣␣
␣␣␣␣###␣␣␣␣
␣␣␣␣###␣###
␣␣␣␣###␣###
###␣###␣###
###␣###␣###
###␣###␣###
###␣###␣###
###␣###␣###
(a)␣(b)␣(c)
```

Note: For clarity, the spaces are explicitly shown in the above example with character (␣).

Hint 1: Several of the functions and boxes you have implemented in the previous tasks may come in handy for this exercise.

Hint 2: You may find the function `intersperse` from the `Data.List` module useful for this exercise.