Haskell Programming Project

Symbolic and Declarative Computing & AI

Binary Image Manipulations

October 17, 2016

The coursework task formulated on the pages below is your Haskell programming coursework. It comprises ten subtasks. I am describing them below and indicating for each subtask (and its components) the maximum number of marks achievable. You are advised to follow the guidance given below. Try to work through the sheet in a systematic fashion (i.e. from beginning to end with as much coverage as possible). Partial solutions will also attract marks if the idea(s) involved show promise or substance.

Individual working and submission are required. Late submission will result in zero marks. You must **staple your submission and number the pages**, following the style of this document, otherwise you risk loosing marks. Typewritten/word processed documents are a *must* in this case as this is the evidence of your software working as required. In addition, you may be asked to demonstrate and explain your software to me in person. For this reason, the submission is not anonymous: **write your name on each page** and staple the pages.

In your report, please reproduce all computer output using a typewriter font. (Use the font `Courier New` if you use Microsoft WORD, and use the `verbatim` environment if you use LaTeX.) The report should also include, as an appendix, your sourcecode in typewriter font. The source code should be presented pleasingly, with self–documenting names for the labels and functions, and you should use comments. The main body of the report should describe each function you have used and the reasons for defining them the way you have. For full marks, not only the software should behave as specified but the report should also be pleasing.

And, to illustrate the working of your functions, you should do this by using examples **different** from the ones used in this sheet. Again, I want to make quite sure that your definitions work as claimed and no 'result' gets pasted in mistakenly from my sheet.

Note. This coursework, as all assessed tasks, has to be seen and approved by the external examiner. For organisational reasons, the process concerned has not been fully completed as yet. BUT, nevertheless, for time reasons and with the

agreement of all concerned, the coursework sheet is being released now, since, as past experience shows, in most cases the suggested coursework sheet passes the scrutiny unchanged or, in some rare instances, with minor additions and/or changes. Should that be the case, the desired changes will be communicated instantly and students will not be disadvantaged thereby.

- **Hand out date:** During week 5 (17 – 21 October 2016) in electronic form on Blackboard.

  This very file is called there `Cw1_16.pdf`. You will also find there the skeleton source code `Cw1_16_skeleton.hs`. Please dowload both files into your filesystem!

- **Submission date:** 5th December 2016 (Monday), by 4.00 pm.

- **Submission to:** Student Support Office, Horton Building as a **hardcopy on paper** ONLY.

- **Items to be submitted:** Please submit *two* hardcopies of your full report (incl. your source code) as specified by the regulations. **DO NOT submit electronic solutions to anyone.**

* * *

Finally, I get asked repeatedly about how coursework marks in general contribute to the assessment of this module. Here it is once more:

- Coursework marks 50% maximum, Exam marks 50% maximum.
- There are three courseworks for this module of which

  #1 Contributes maximum 25 % (**This very coursework!**)

  #2 Contributes maximum 12.5 %,

  #3 Contributes maximum 12.5 %.

* * *

Context for the Curious Student

In this coursework sheet, you are asked to implement in Haskell some simple image manipulations for *binary* images. Our module is, of course, not about Image Processing. However, I believe it is of educational value to interrelate our skills to areas of possible application, one of them being binary image processing.

The following is an interesting reference:

R. Jain, R. Kasturi and B. G. Schunk, *Machine Vision*, McGraw-Hill, New York, 1995.

You may find in particular of interest, Chapter 2 entitled "Binary Image Processing", pages 25–72.

I hasten to add that the information in this book is not part of the material covered or examined; it is purely additional background information for the curious student.

The Tasks

In Outline

You are asked to implement in Haskell a toolbox for manipulating *binary images*. These will be represented using constants of the type `[String]`. The initial part of my program looks like this:

```
module Cw2016 where
import Char
import Hugs.Prelude

type BPic = [String]

bpic :: Int -> BPic
bpic 1 = [" XXX ",
          "X   X",
          "XXXXX",
          "X   X",
          "X   X"]

bpic 2 = [" XXX ",
          "X   X",
          "XXXXX",
          "X X",
          "X   X"]

bpic 3 = ["X      XXX   XX   XX  XXX ",
          "X     X   X  X X X X  X  X",
          "X     XXXXX  X  X  X  XXX ",
          "X     X   X  X     X  X   ",
          "XXXXX X   X  X     X  X   "]

bpic 4 = ["    XXXXXXX XX ",
          "  XXXXXXXXXXXXX",
          "    XXX XXXXXXX",
          "    XXX     XXX",
          "X   XXX      XX",
          "X   XX    X  XX",
          "X   XX   XX  X ",
          "    XX   XX  X ",
          "    XXXXXXX    ",
          "    XXXXXXX    ",
          "    XXX XXX    ",
          "    XX   XX  XX",
          "    XXX XXX  XX",
          "    XXX  XX  XX",
          "    XX   XX  XX",
```

```
         "    XXX      XX",
         "    XXX      XX",
         "  XXXXXXXXXXXX",
         " XXXXXXXXXXXXX",
         "  XXXXXXXXXX X"]



bpic 5 = ["               ",
          "      XX       ",
          "    XXXXXX     ",
          "  XXXXXXXXXX   ",
          "XXXXXXXXXXXXX ",
          "  XXXXXXXXXX   ",
          "    XXXXXX     ",
          "      XX       ",
          "               ",
          "               ",
          "               "]

bpic 6 = ["               ",
          "               ",
          "               ",
          "               ",
          "      XX       ",
          "    XXXXXX     ",
          "  XXXXXXXXXX   ",
          "XXXXXXXXXXXXX ",
          " XXXXXXXXXX    ",
          "  XXXXXX       ",
          "   XX          "]
```

To save you some typing work, this skeleton of the source I am making available in Blackboard as `Cw1_16_skeleton.hs`. Here it is how to invoke it interactively and what you will see on the screen:

```
[acsenki@d40601] Cw16 #: hugs Cw1_16_skeleton
__   __ __  __  ____   ___      ----------------------------------------
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 ----------------------------------------

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Cw2016>
```

Above I define six binary images (of various sizes). (To be more precise, image number 2 won't qualify as such since one of its rows (of type `String`)

is shorter than the other ones.) Anyway, the above illustrates how I chose to represent images.

You should define your own pictures by adding further to the definition of the function `bpic`. These pictures are very simple: each pixel ('picture element') is either white (indicated by a space in our model) or it is black (indicated by `X` in our model). What can be done to such images? A great deal. The following is a list of what should be considered by you in this coursework.

(1) They can be *displayed* on the terminal, *checked* whether they represent a picture in the first instance, and, their *size* can be determined.

(2) A simple *set theoretic operation* on a binary image is that of *complementation*: black pixels become white and vice versa.

(3) Another two *set theoretic operations*, now carried out on a pair of binary images, are the operations *union* and *intersection*.[1]

(4) Pictures can be *rolled up* or *rolled down vertically* by one row each or by a specified number of rows. (Vertical 'wrapping' is applied – I will explain details later.)

(5) Pictures can be *moved horizontally* in the same way as described in the previous point. (Now, horizontal 'wrapping' is applied – as before, I will explain details of this later.)

(6) Combining the two actions described in (4) and (5), we can subject the image to a *directed movement*. (I shall call this a *shift*.) (Again, with 'wrapping'.)

(7) Whereas in (4)–(6), I was assuming that the image content is not lost (e.g. when rolling up, the earlier top line will now appear at the bottom), we can essentially 'redo' the whole exercise (4)–(6) by now assuming that *empty cells* are reintroduced, i.e. *without* wrapping. (The idea will be clear when illustrating it later.)

(8) *Framing* images as demonstrated later.

(9) *Mirroring* images as demonstrated later.

(10) *Rotating* images as demonstrated later.

Details, Guidance and Marking Scheme

Your tool should be working in the described manner for *any* binary images of the kind described above. Therefore:

In your submission, please provide evidence that your tool is working also for some other binary images of your own creation in the manner described here.

[1] These two operations and the one described in the previous point is akin to operations usually performed on *Venn diagrams*.

Below I describe in detail ten tasks and tell you the maximum number of marks you can collect for each of them. The maximum achievable grand total is 100. I will convert the total number of marks achieved by you to a percentage and record that figure. Submissions attracting 70 marks or more will be awarded 100%. The percentage will be the *smaller* of the numbers 100 and total number of marks achieved divided by 0.7, i.e.

$$\text{percentage recorded} = \text{minimum}\{100, \frac{\text{total number of marks achieved}}{0.7}\}$$

This means in effect that any marks beyond 70 will be 'discarded' by me. Nevertheless, you are encouraged to solve as many questions as you can for three reasons:

- This coursework should be seen as an *opportunity* to develop and practice your Haskell skills. The skills thus acquired will be useful also in the *examination*.
- The number of marks indicated next to each task is a *maximum*. Marks may be lost for example for inadequate reporting. It is therefore *safer* to be aiming for more than 70 marks.
- And, finally, I hope that you will derive some *pleasure* from this coursework as it invites you to be *creative*.

The solution of a task may help in solving another task, though you may attempt tasks in any order. Nevertheless, you are adviced to try them consecutively.

(1) Three functions should be defined here.

- Define `showPic :: BPic -> IO ()` for displaying pictures on the terminal. Example:

```
Cw2016> showPic (bpic 3)
X      XXX   XX   XX  XXX
X     X   X  X X X X  X  X
X     XXXXX  X  X  X  XXX
X     X   X  X     X  X
XXXXX X   X  X     X  X
```

- Furthermore, define `sizePic :: BPic -> (Int, Int)` for finding out the size of a picture as demonstrated here.

```
Cw2016> sizePic (bpic 3)
(26,5)
```

- Finally, define `isPic :: BPic -> Bool` for checking whether an item of type `BPic` is a binary picture. Examples:

```
Cw2016> isPic (bpic 3)
True
Cw2016> isPic (bpic 2)
False
```

Suggestion. I found it useful in my implementation to define and use here the auxiliary function `isEqual :: [Int] -> Bool`; example:

```
Cw2016> isEqual [8,8,8,8]
True
Cw2016> isEqual [8,8,4,8]
False
```

You may wish to define and use `isEqual` in your implementation of `isPic` too.

The maximum achievable number of marks for each task is 7, making a total of 21. *marking advice*

(2) Define `compPic :: BPic -> BPic` for picture complementation. Example:

```
Cw2016> showPic (bpic 5)

      XX
    XXXXXX
  XXXXXXXXXX
XXXXXXXXXXXXX
  XXXXXXXXXX
    XXXXXX
      XX



Cw2016> showPic (compPic (bpic 5))
XXXXXXXXXXXXXX
XXXXXX  XXXXXX
XXXX      XXXX
XX          XX
              X
XX          XX
XXXX      XXXX
XXXXXX  XXXXXX
XXXXXXXXXXXXXX
XXXXXXXXXXXXXX
XXXXXXXXXXXXXX
```

Suggestion. You may wish to start by implementing the same operation on one-dimensional binary 'pictures' represented simply by (binary) lists. This would mean defining a Haskell function `compL :: String -> String`, say, which then would behave like this:

```
Cw2016> compL [' ',' ','X','X',' ']
"XX  X"
```

Now use `compL` in conjunction with the built-in function `map` to define `compPic`.

The maximum achievable number of marks for this task is 7. *marking advice*

(3) Define `unionPic, isecPic :: BPic -> BPic -> BPic` for respectively creating the union and intersection of two binary pictures. (Naturally, they have to be of the same size.)[2]

Example: do these operations on picures five and six. Start by showing picture six (as we have seen picture five already).

```
Cw2016> showPic (bpic 6)
```

```
      XX
    XXXXXX
  XXXXXXXXXX
XXXXXXXXXXXXX
 XXXXXXXXXX
  XXXXXX
   XX
```

Now, do the operations union and intersection :

```
Cw2016> showPic (unionPic (bpic 5) (bpic 6))
```

```
      XX
    XXXXXX
  XXXXXXXXXX
XXXXXXXXXXXXX
  XXXXXXXXXX
  XXXXXXXXXX
XXXXXXXXXXXXX
 XXXXXXXXXX
  XXXXXX
   XX
```

```
Cw2016> showPic (isecPic (bpic 5) (bpic 6))
```

```
      XX
    XXXXXX
    XXXXXX
      XX
```

Suggestion. You may wish to start by implementing the same operations on one-dimensional binary 'pictures' represented simply by (binary)

[2]Union and intersection are defined pixel–wise. That is, a pixel at a given position in the *union* picture is black if the same position is black *in at least one* of the input pictures, white otherwise. And, a pixel at a given position in the *intersection* picture is black if the same position is black *in both* input pictures, white otherwise.

lists. This would require, for example for the union operation, defining a Haskell function `unionL :: String -> String -> String`, say, which then would behave like this:[3]

```
Cw2016> unionL [' ',' ','X','X',' '] ['X',' ','X',' ','X']
"X XXX"
```

Now use `unionL` in conjunction with the built-in function `zipWith` to define `unionPic`. Define `isecL` along similar lines.

Alternatively, you may wish to define `isecL` by using `compPic` from (2) and `unionPic`, based on the set theoretic identity

$$A \cap B = (A^c \cup B^c)^c.$$

This definition will be very concise, just one line. Do both definitions if possible.

The maximum achievable number of marks for each task is 7, making a total of 14. *marking advice*

(4) Define `rollup :: BPic -> BPic` for 'rolling' the picture up *one row*. Example:

```
Cw2016> showPic (rollup (bpic 3))
X     X   X  X X X X  X  X
X     XXXXX  X  X  X  XXX
X     X   X  X     X  X
XXXXX X   X  X     X  X
X      XXX   XX   XX  XXX
```

(Similarly, define `rolldown`.) (Notice that the impression is created that the text is moving up and down a tube!) For a given number of lines of rolling, `rollPic :: Int -> BPic -> BPic` 'rolls' the argument picture. The amount of rolling is specified by the first argument. Example:[4]

```
Cw2016> showPic (rollPic (-2) (bpic 3))
X     X   X  X     X  X
XXXXX X   X  X     X  X
X      XXX   XX   XX  XXX
X     X   X  X X X X  X  X
X     XXXXX  X  X  X  XXX
```

The maximum achievable number of marks for completing this task is 14. *marking advice*

(5) This task is in a sense an *analogue* of task (4) in that we consider here moving the picture horizontally. However, the task is probably more complicated due to the way the picture is internally represented row–wise.

Now, define `moveright :: BPic -> BPic` as seen here.

[3] *Suggestion.* Use the built-in function `zipWith` to define `unionL`.

[4] The first argument of `rollPic` is negative if wanting to roll down, positive otherwise.

```
Cw2016> showPic (moveright (bpic 1))
  XXX
XX
XXXXX
XX
XX
```

(Observe again the impression of folding around a tube, now vertical.) The function `moveleft :: BPic -> BPic` should be defined such that it does the analogous thing.

The third function `movePic :: Int -> BPic -> BPic` is to be defined using the former two; it moves a picture horizontally a specified number of cells in the specified direction. Example:

```
Cw2016> showPic (movePic (-5) (bpic 3))
  XXX   XX   XX  XXX X
 X   X  X X X X  X  XX
 XXXXX  X  X  X  XXX X
 X   X  X     X  X   X
 X   X  X     X  X   XXXXX
```

Suggestion. I have found it useful in my implementation to define and use here the auxiliary function `fusePic :: BPic -> BPic -> BPic`;[5] it 'fuses' two images to one new one as demonstrated below:

```
Cw2016> showPic (fusePic (bpic 1) (bpic 3))
 XXX X      XXX   XX   XX  XXX
X   XX     X   X  X X X X  X  X
XXXXXX     XXXXX  X  X  X  XXX
X   XX     X   X  X     X  X
X   XXXXXX X   X  X     X  X
```

It is simple and short to define `fusePic`; just use the built-in function `zipWith` in conjunction with the concatenation operator `++`.

In this task, I have used the buil-in functions `map`, `head`, `last`, `init`, `zipWith` and `++`.

The maximum achievable number of marks for completing this task is 14. *marking advice*

(6) Now define `shiftPic :: Int -> Int -> BPic -> BPic` for shifting a picture as specified by the first two arguments. As an example, we shift picture five by three cells to the right and by four cells up. This is illustrated by the three consecutive pictures shown below.

[5] The way I have achieved moving one step to the left, say, is as follows. I use `map` to create from a given picture two slimmer pictures: I take the first 'vertical slice' of the given picture and then I take the rest and now fuse them together by appending the first slice to the second.

```
Cw2016> showPic (shiftPic 0 0 (bpic 5))

      XX
    XXXXXX
  XXXXXXXXXX
XXXXXXXXXXXXX
  XXXXXXXXXX
    XXXXXX
      XX



Cw2016> showPic (shiftPic 3 0 (bpic 5))

         XX
       XXXXXX
     XXXXXXXXXX
XX XXXXXXXXXXX
     XXXXXXXXXX
       XXXXXX
         XX



Cw2016> showPic (shiftPic 3 4 (bpic 5))
XX XXXXXXXXXXX
     XXXXXXXXXX
       XXXXXX
         XX




         XX
       XXXXXX
     XXXXXXXXXX
```

Suggestion. The function `shiftPic` is easily (and concisely) defined in terms of the functions `rollPic` and `movePic` defined in (4) and (5) respectively.

The maximum achievable number of marks for this task is 5. *marking advice*

(7) This is a variant of (4)–(6). Now define a function

```
shiftPic_e :: Int -> Int -> BPic -> BPic
```

for shifting a picture as specified by the first two arguments, and now do **not** 'wrap' the picture around. As an example, we do the same operations as in (6); this is illustrated below.

```
Cw2016> showPic (shiftPic_e 3 4 (bpic 5))
```

```
 XXXXXXXXXXXX
   XXXXXXXXXX
     XXXXXX
       XX
```

Another example which illustrates the point once more is:

```
Cw2016> showPic (shiftPic_e 0 0 (bpic 3))
X      XXX   XX   XX  XXX
X     X   X  X X X X  X  X
X     XXXXX  X  X  X  XXX
X     X   X  X     X  X
XXXXX X   X  X     X  X

Cw2016> showPic (shiftPic_e (-5) (-1) (bpic 3))

  XXX   XX   XX  XXX
 X   X  X X X X  X  X
 XXXXX  X  X  X  XXX
 X   X  X     X  X
```

Suggestion. I have done the following. I have inspected my solutions to (4)–(6) and made the requisite changes to my function definitions.[6] The changes are relatively minor.

The maximum achievable number of marks for this task is 5. *marking advice*

(8) Define `framePic :: BPic -> BPic` for framing pictures. Example:

```
Cw2016> showPic (framePic (bpic 3))
+---------------------------+
| X      XXX   XX   XX  XXX  |
| X     X   X  X X X X  X  X |
| X     XXXXX  X  X  X  XXX  |
| X     X   X  X     X  X    |
| XXXXX X   X  X     X  X    |
+---------------------------+
```

Notice that the border comprises two 'layers': (i) the empty layer around the original picture, and (ii) the lines around it.

Suggestion. There are various ways of doing this. As I had the function `fusePic` at this stage available, I thought I will make use of it here. Then, the solution may go like this.

[6] For example, horizontal movement now involves removing the first column (say) of a picture and then appending to what is left an 'empty image' comprising a single column of spaces. Being able to fuse pictures is now handy!

❶ Define `filledL :: Char -> Int -> String` for creating a list of a specified length with each list element being the same character. Example:

```
Cw2016> filledL '|' 10
"||||||||||"
```

Use *recursion* here.

❷ Define `filledPic :: Char -> (Int,Int) -> BPic` for creating a picture of a specified size with each pixel containing the same character. Example:

```
Cw2016> showPic (filledPic '|' (10, 3))
||||||||||
||||||||||
||||||||||
```

Use `filledL` in the definition of `filledPic` and define `filledPic` by *recursion*.

❸ Define `framePic_v :: BPic -> BPic` for 'framing' a picture partially, from the sides, i.e. vertically. Example:

```
Cw2016> showPic (framePic_v (bpic 3))
| X      XXX   XX   XX  XXX  |
| X     X   X  X X X X  X  X |
| X     XXXXX  X  X  X  XXX  |
| X     X   X  X     X  X    |
| XXXXX X   X  X     X  X    |
```

Use here `filledPic` from ❷ and `fusePic`. (The latter was discussed in (5)). The idea here is to 'fuse' the original picture with two new pictures, the borders, each of width two.[7] The first one will serve as the left hand border and it is itself obtained by fusing two pictures each of width one. Therefore, fusing will be applied several times. I won't pursue this further here as the idea should be clear by now.

❹ We are almost there! The top and bottom borders remain to be done. (Measure now the width of the picture with its sideframes and add four lists to its representation: two to the front, two to the end. (The latter operation involves concatination, `++`.)

Notice. The above suggestion describes a systematic approach and the availability of some previously defined functions. **However, you may alternatively answer this question independently of all the others in an ad hoc manner.** After all, framing of pictures, if represented in the way done here, can be achieved by using the built-in function `map`. Such an implementation will be very concise and can be tried out interactively.

The maximum achievable number of marks for this task is 7. *marking advice*

[7] Needless to say, we will have to measure the size of the original picture so as to match the border size; this is done (inside `framePic_v`) by using `sizePic` from (1).

(9) Now define `mirror_h, mirror_v :: BPic -> BPic`, functions for mirroring a binary picture horizontally and vertically, respectively. Example:

```
Cw2016> showPic (mirror_h (bpic 3))
XXXXX X   X  X     X  X
X     X   X  X     X  X
X     XXXXX  X  X  X  XXX
X     X   X  X X X X  X  X
X      XXX   XX   XX  XXX

Cw2016> showPic (mirror_v (bpic 3))
 XXX  XX   XX   XXX      X
X  X  X X X X  X   X     X
 XXX  X  X  X  XXXXX     X
   X  X     X  X   X     X
   X  X     X  X   X XXXXX
```

Suggestion. These functions are simple to define. Use the built-in functions `reverse` and `map`. The definitions won't use any previously defined function.

The maximum achievable number of marks for this task is 7. *marking advice*

(10) Rotating binary images. This is in several steps.

(i) Define `rotOnce :: BPic -> BPic` for rotating a binary image quarter–turn counterclock–wise. Example:

```
Cw2016> showPic (rotOnce (bpic 3))
 X
X X
X X
XXXXX


XXXXX
X
 X
  X
 X
X
XXXXX


 XXXX
X X
X X
X X
 XXXX

    X
    X
    X
    X
XXXXX
```

Suggestion. This *may appear* a rather difficult task but I am going to guide you!

❶ Define `moveCol :: (BPic, BPic) -> (BPic, BPic)`, a function for taking a pair of pictures and 'slicing off' the first column of the first picture and 'putting it on top' of the second one. The following example demonstrates what happens if I *repeatedly* apply this function.

```
Cw2016> bpic 1
[" XXX ","X   X","XXXXX","X   X","X   X"]
Cw2016> moveCol (bpic 1, [])
(["XXX ","   X","XXXX","   X","   X"],[" XXXX"])
Cw2016> moveCol $$
(["XX ","  X","XXX","  X","  X"],["X X  "," XXXX"])
Cw2016> moveCol $$
(["X "," X","XX"," X"," X"],["X X  ","X X  "," XXXX"])
Cw2016> moveCol $$
([" ","X","X","X","X"],["X X  ","X X  ","X X  "," XXXX"])
Cw2016> moveCol $$
(["","","","","",""],[" XXXX","X X  ","X X  ","X X  "," XXXX"])
Cw2016> moveCol $$
(["","","","","",""],[" XXXX","X X  ","X X  ","X X  "," XXXX"])
Cw2016> showPic (snd $$)
 XXXX
X X
X X
X X
 XXXX
```

❷ The above interactive session shows that repeatedly applying `moveCol` eventually results in the rotated image. This observation suggests using the built-in function `iterate`. The seed is also suggested here. (Look at closely the interactive session shown in ❶!) We define therefore `picPairs` by

```
picPairs :: BPic -> [(BPic, BPic)]
picPairs pic = iterate moveCol (pic, [])
```

It will produce an infinte list of picture pairs. Try it out!

❸ Using `iterate` as in ❷ will produce an infinite list but we want only *one* entry of that infinite list,[8] namely the one to which the infinite list eventually settles.

Define now `rotOnce :: BPic -> BPic` for rotating a picture once as demonstrated earlier. Use the infinite list produced by `picPairs` in ❷ and also use the built-in functions `dropWhile`, `snd` and `head`.

The max. achievable number of marks for this task is 4. *marking advice*

(ii) Define `rotPic :: Int -> BPic -> BPic` for rotating a binary image quarter–turn a specified number of times. (Allow for the *possibility* that the first argument of `rotPic` is negative. Then, we mean rotation clockwise.) Example:

```
Cw2016> showPic (rotPic 2 (bpic 3))
   X  X     X  X   X XXXXX
   X  X     X  X   X     X
 XXX  X  X  X  XXXXX     X
X  X  X X X X  X   X     X
 XXX  XX   XX   XXX      X
```

[8] I mean really the second component of that entry as the entry is a pair.

Suggestion. Use `rotOnce` from (i) and define `rotPic` by recursion.
The max. achievable number of marks for this task is 2. *marking advice*