University of Arizona, Department of Computer Science

CSc 372 — Assignment 3 — Due noon, Wed Oct 12 — 8%

Christian Collberg
September 23, 2005

1 Introduction

The purpose of this assignment is to work more with higher-order functions.

Every function you write for this assignment (except when explicitly noted) should be *non-recursive.* I.e. functions will typically be implemented using higher-order functions such as `maps`, `folds`, `zips`, etc.

You may freely introduce auxiliary functions if that makes your program cleaner. Also, feel free to introduce local definitions (`where`-clauses) to make your code easier to read.

You will be graded primarily on correctness and style, not on the execution efficiency of your code.

All functions must be commented.

The first half of the assignment (**A**) you should solve individually. The second part (**B**) you can work on in teams of two.

You should hand in three files (`ass3A.hs ass3B.hs TEAM`), one for each part, and one file that lists the names and logins of the students in your team. Only one team member needs to hand in `ass3B.hs`.

2 A: Individual Problems

1. Write a function `maxl xs` that generates an error `"empty list"` if `xs==[]` and otherwise returns the largest element of `xs`: [10 points]

```
maxl :: (Ord a) => [a] -> a
maxl xs = ...

> maxl [2,3,4,5,1,2]
5
> maxl []

Program error: empty list
```

2. Write a function `mull xs m` which returns a new list containing the elements of `xs` multiplied by `m`: [10 points]

```
mull :: (Num a) => [a] -> a -> [a]
mull xs x = ...
```

```
> mull [1,2,3,4,5] 0.0
[0.0,0.0,0.0,0.0,0.0]
> mull [1,2,3,4,5] 2
[2,4,6,8,10]
> mull [2.0,4.0,6.0,8.0,10.0] 5
[10.0,20.0,30.0,40.0,50.0]
```

3. Write a function `member x ys` which returns `True` if `x` is an element of `ys`, `False` otherwise. [10 points]

```
member :: (Eq a) => [a] -> a -> Bool
member xs s = ...

> member [1,2,3] 4
False
> member [1,2,3,4.0,5] 4
True
> member ['a','b','x','z'] 'b'
True
```

4. Write a function `setsub xs ys` that takes two lists `xs` and `ys` as input, both lists representing sets. In other words, `xs` and `ys` are unsorted lists that contain no duplicate elements. `setsub xs ys` returns `xs-ys`, i.e. the list containing the elements in `xs` that are not in `ys`: [10 points]

```
setsub :: (Eq a) => [a] -> [a] -> [a]
setsub xs ys = ...

> setsub [1,2,3] []
[1,2,3]
> setsub [1,2,3] [3]
[1,2]
> setsub [1,2,3] [1,2,3]
[]
> setsub [] [1,2,3]
[]
> setsub [] []
[]
> setsub [True,False] [True]
[False]
```

5. In the definition of `sumc` below, use *function composition* to compute the sum of the cubes of all the numbers divisible by 7 in a list `xs` of integers. [10 points]

```
sumc :: [Int] -> Int
sumc =
   where cube x = ...
         by7  x = ...

> sumc [7]
343
> sumc [7,14]
```

```
3087
> sumc [7,8,14]
3087
```

Note: In this case, I want the definition of `sumc` to look exactly as above.

3 B: Pixel Displays

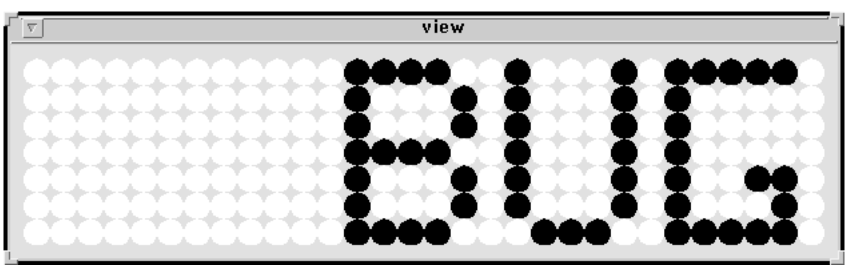
Pixel displays are seen on busses, in airports, in shop windows, etc. They are often made up of rows of LEDs (Light Emitting Diodes), and display their messages rotating, scrolling, flashing, etc.

To get an idea of what a pixel display looks like, download the files `pixels1.txt,···,pixels5.txt` and `view.icn` from the course web-site: `http://www.cs.arizona.edu/~collberg/Teaching/372/2005/Assignments`. Then do the following:

```
> icont view.icn
> view pixels1.txt 30 100
> view pixels2.txt 30 100
...
```

The `pixels*.txt`-files show how a text should scroll, invert, etc. The `view` program is an Icon program that simulates a pixel display. It takes a `pixels*.txt` file as input as well as two parameters *delay* and *repeat* which sets the delay (in micro-seconds) and the number of times to repeat, respectively.

The display will look something like this:



In this assignment you will write a Haskell program that takes a specification for how a text should be scrolled, etc, as input and produces a `pixels*.txt` file as output.

3.1 Templates

To start off, you can pick up a Haskell template `ass3B-template.hs` from `http://www.cs.arizona.edu/~collberg/Teaching/372/2005/Assignments`. It contains, among other things, a function `main s ops` that takes two arguments: a string `s` to be displayed and a list of functions `ops` (for scrolling, inverting, etc) to be applied to the display. `main` will produce a string showing how the display is updated at each step. Here's an example where `main` produces a pixel display for the string `"BUG"`, and then rotates it left twice:

```
? main "BUG" [left,left]
****  *   * *****
*   * *   * *
*   * *   * *
****  *   * *
*   * *   * *  **
*   * *   * *   *
****   ***  *****

***  *   * ***** *
   * *   * *     *
   * *   * *     *
***  *   * *     *
   * *   * *  ** *
   * *   * *   * *
***   ***  ***** *

**  *   * ***** **
  * *   * *     *
  * *   * *     *
**  *   * *     **
  * *   * *  ** *
  * *   * *   * *
**   ***  ***** **
```

Your task is to fill in the missing function definitions in the template.

3.2 Fonts

We start out be defining a type `Pixels`, which is a list of `String`, or, equivalently, a list of lists of `Char`s. Objects of type `Pixels` will be used to build up our display.

```
type Pixels = [String]
```

Next, we define the “font”-description which gives the layout of each character of the alphabet:

`font ::  Char -> Pixels`

`font 'A' = ["␣***␣",`	`font 'B' = ["****␣",`	`font ' ' = ["␣␣␣␣␣",`
`"*␣␣␣*",`	`"*␣␣␣*",`	`"␣␣␣␣␣",`
`"*␣␣␣*",`	`"*␣␣␣*",`	`"␣␣␣␣␣",`
`"*****",`	`"****␣",`	`"␣␣␣␣␣",`
`"*␣␣␣*",`	`"*␣␣␣*",`	`"␣␣␣␣␣",`
`"*␣␣␣*",`	`"*␣␣␣*",`	`"␣␣␣␣␣",`
`"*␣␣␣*"]`	`"****␣"]`	`"␣␣␣␣␣"]`

An asterisk (`'*'`) represents a black pixel, and a blank (`'␣'`) represents a white. An incomplete font definition (upper case characters only) can be found in the template.

3.3 Printing Pixels [10 points]

Our Haskell program will communicate with the display unit itself via text files. We therefore need to be able to convert the pixel representation into the equivalent strings. `pixelsToString` converts a `Pixel`-object $[s_1, s_2, \cdots, s_n]$ (a list of `String`s) to a string by appending $s_1, \cdots, s_n$ together with newlines (`\n`) in between: "s_1`\n`s_2`\n`$\cdots$`\n`s_n`\`". Here is an example:

```
> pixelsToString (font 'A')
" *** \n*   *\n*   *\n*****\n*   *\n*   *\n*   *\n\n"
> putStr (pixelsToString (font 'A'))
 ***
*   *
*   *
*****
*   *
*   *
*   *
```

The function `pixelListToString` is similar. It takes a list of `Pixels` as argument, converts each argument to a string (using `pixelsToString`), and then appends the strings together with an extra newline in between:

```
> pixelListToString [font 'A', font 'B']
" *** \n*   *\n*   *\n*****\n*   *\n*   *\n*   *\n\n
\n**** \n*   *\n*   *\n**** \n*   *\n*   *\n**** \n\n\n\n"
> putStr (pixelListToString [font 'A', font 'B'])
 ***
*   *
*   *
*****
*   *
*   *
*   *


****
*   *
*   *
****
*   *
*   *
****
```

(NOTE: In the first command above I had to split the result over two lines not to exceed the line length of this page. Your function would put all its result on one line.)

Give definitions of `pixelsToString` and `pixelListToString` according to the signatures and examples below. For clarity, the function results are given twice: both the way they will be printed on the screen (left) and with newlines and spaces explicit (right). Remember that your function definitions should not use recursion, just higher-order functions such as `foldr, foldl, zip, map`, etc.

3.4 Appending Pixels [10 points]

Next, we need to define operations to compose larger pixel displays from smaller ones. We start by defining an operation `appendPixels xs ys` which puts two pixels-objects together horizontally:

```
                 appendPixels ::  Pixels -> Pixels -> Pixels
pixelsToString(appendPixels(font 'A')(font 'B')) | appendPixels(font 'A')(font 'B')
-------------------------------------------------+---------------------------------
               *** ****                          |     ["␣***␣****␣",
              *   **   *                         |      "*␣␣␣**␣␣␣*",
              *   **   *                         |      "*␣␣␣**␣␣␣*",
              *********                          |      "*********␣",
              *   **   *                         |      "*␣␣␣**␣␣␣*",
              *   **   *                         |      "*␣␣␣**␣␣␣*",
              *   *****                          |      "*␣␣␣*****␣"]
```

The `concatPixels` function is similar to `appendPixels` but takes a list of `Pixels` as argument. The `messageToPixels` function (which has already been defined for you), finally, converts a message (a string of uppercase letters) into a pixel object, inserting an extra space between each character. Here are the signatures and some examples:

```
     concatPixels ::  [Pixels] -> Pixels         | messageToPixels ::  String -> Pixels
pixelsToString(concatPixels[font 'A',font 'B'])  | pixelsToString(messageToPixels "BUG")
-------------------------------------------------+---------------------------------------
               *** ****                          |      ****  *   * *****
              *   **   *                         |      *   * *   * *
              *   **   *                         |      *   * *   * *
              *********                          |      ****  *   * *
              *   **   *                         |      *   * *   * *  **
              *   **   *                         |      *   * *   * *   *
              *   *****                          |      ****   ***  *****
```

3.5 Pixel Operations [15 points]

Next, we define the operations we want to be able to perform on the display. The most common operations are rotation (left, right, up, and down) and inversion (changing all white pixels to black and vice versa). We can also flip a picture upside-down, or turn it backwards. For example, here are the results of rotating the character `'A'` up two pixels, rotating the character `'B'` left one pixel, flipping an `'A'` upside-down, and turning a `'B'` backwards:

`pixelsToString (up (up (font 'A')))`	`pixelsToString (left (font 'B'))`
`*   *`	`*** *`
`*****`	`   **`
`*   *`	`   **`
`*   *`	`*** *`
`*   *`	`   **`
` ***`	`   **`
`*   *`	`*** *`
`pixelsToString (upsideDown (font 'A'))`	**`pixelsToString (backwards (font 'B'))`**
`*   *`	` ****`
`*   *`	`*   *`
`*   *`	`*   *`
`*****`	` ****`
`*   *`	`*   *`
`*   *`	`*   *`
` ***`	` ****`

All pixel operations have the type `Pixels -> Pixels`:

```
type PixelOp = Pixels -> Pixels
up, down, left, right, invert, upsideDown, backwards ::  PixelOp
```

i.e. they convert `Pixels` into `Pixels`. Your task is to provide *non-recursive* definitions of `invert, up, down, left, right, upsideDown`, and `backwards`. It is suggested that you use the functions `map, ++, init, last, head, tail`, and `reverse` from the standard prelude.

3.6 Main Functions [10 points]

Now we're ready to put it all together. The function `main s fs` defined below takes two arguments as input: a string `s` to be displayed and a list of functions `fs` to be applied to the display. `mainFile` performs the same operation as `main` but writes the result to a file.

```
main ::  String -> [PixelOp] -> String
main s fs = pixelListToString (appColl (messageToPixels s) (id:fs))

mainFile ::  String -> String -> [PixelOp] -> Dialogue
mainFile file s fs = writeFile file (main s fs) abort done
```

All that's left (for you) to do is to provide a definition of the *apply-collect*-function `appColl  pixels ops` function. It takes two arguments: the first argument is the pixel-object to be manipulated, the second argument is a list of operations (such as `invert, up, down`, etc.) that are to be applied to the object. The result is a list of `Pixels` representing all the updates that have to be performed on the display. Hence, `appColl` is a higher-order function with this signature:

```
appColl ::  Pixels -> [PixelOp] -> [Pixels]
appColl pixels ops = ···
```

We can express the functionality of `appColl` a bit more formally like this:

```
appColl p [f₁,···,fₙ] = [
   f₁ p,
   f₂(f₁ p),
   f₃(f₂(f₁ p)),
         ···
   fₙ(fₙ₋₁(···(f₂(f₁ p))···))
]
```

$$\texttt{appColl p [}f_1,\cdots,f_n\texttt{] = [}\; f_1\ p,\; f_2(f_1\ p),\; f_3(f_2(f_1\ p)),\; \ldots,\; f_n(f_{n-1}(\cdots(f_2(f_1\ p))\cdots))\;\texttt{]}$$

As a final example, assume that we're starting out with a pixel-object `p`. We can then use `appColl` to compute the result (and all intermediate results) of scrolling to the left twice, and up once:

```
appColl p [left,left,up] ⇒ [left p, left(left p), up(left(left p))]
```

You may use recursion to define `appColl`.

3.7 Application [5 points]

Write a 0-argument function `cool` that displays a message (such as your name) using a combination of effects. Feel free to implement your own special effects, in addition to the ones (`up`, `invert`, etc.) we've already defined.

4 Submission and Assessment

The deadline for this assignment is noon, Wed Oct 12. It is worth 8% of your final grade.

You should submit the assignment electronically using the `Unix` command

`turnin cs372.3 ass3A.hs ass3B.hs TEAM`.

Don't show your code to anyone, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the instructor or the TA.