Exam 1: Final Exam

Due: 11PM, Dec. 11, 2017

Contents

General Instructions

First, download the support code `final.zip`, which can be found at `/course/cs0170/src/final/final.zip`. Unzip it to find to find eight files, one for each of the first four problems, two for the last, a README, and `CS17setup.ml`. Do your work in the appropriate file for each problem, being certain not to alter any portion of the file except the parts indicated. Then re-zip things, and submit via the web-handin mechanism. *Be sure to check the email that comes back to you to be certain that everything you submitted was accepted; be sure to open to attached zip file to be sure the contents are what you expected, and that you did not accidentally submit empty solutions, for instance.* It's up to you to check this; we won't accept late handins with excuses about messing up on the technology.

All coding should follow the CS17 standards, and use the design recipe as we've done all semester.

If you have a question about the exam, email the TA list. Updates and clarifications about the exam will be posted via Piazza.

1 This, that, or the other. (OCaml, 10 points)

You're given a list of data, and a function that for each datum returns the value -1, 0, or $+1$; you must produce a triple of lists (p, q, r) where p consists of all items for which the value is -1; q consists of all those for which the value is 0, and r consists of all those for which the value is $+1$; within each list, items must be in their original order. For instance, if the function was

```
let sign (n: int) = compare n 0;;
```

and the input list was `[1; 0; -2; 4; 2; -1; -6; 0; 0; 3]` then the output would be the triple `([-2; -1; -6],[ 0; 0; 0],[1; 4; 2; 3])`.

Write a procedure `ttoto` that does this. Make sure it can operate on any function, like sign, that matches the description above. Test your procedure using `sign` and at least one other input procedure of your own creation. Be sure to test and comment any procedures you write. The full design recipe and code for `sign` is contained in the support code file for this problem.

2 Waterfall folder displays (OCaml, 25 points)

For this problem, you'll write a procedure that produces an textual representation of a directory's structure.

The Linux operating system organizes files into a tree-like hierarchy. Directories contain files, some of which are themselves directories, and others of which are "plain files". To "list" a directory is to provide a list of the names of all the files it contains, including those that are directories (which are called "subdirectories"). To "recursively list" a directory is to show what's contained within each subdirectory, and each sub-sub-directory, and so on. For instance, if the directory "food" contains "lemonade", "veggies", "dairy", "grains", and "twizzlers", and "dairy" is a directory containing nothing, "grains" is a directory containing "oats" and "quinoa", and "veggies" contains "celery" and "potato", the the linux command "ls -R" (recursively list directory contents) produces this:

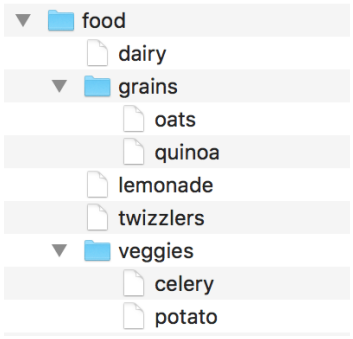
```
% ls -R food
dairy      grains       lemonade.txt    twizzlers.txt   veggies

food/dairy:

food/grains:
oats.txt    quinoa.txt

food/veggies:
celery.txt  potato
```

That's not tough to read, but it's not as nice as what a Mac provides:



We're going to compromise, and write a procedure, `waterfall`, that takes in a file and produces a textual representation of that file that's similar to the Mac picture, and has a cascading appearance, hence the name "waterfall." For the directory structure above, the output we want looks like this:

```
food
|-- dairy
|-- grains
|   |-- oats
|   |-- quinoa
|-- lemonade
|-- twizzlers
|-- veggies
    |-- celery
    |-- potato
```

A listing of a more complex structure might look like this (while being a bit messed up as far as food categories go!):

```
food
|-- dairy
|-- grains
|   |-- eastern
|   |   |-- bulgur
|   |   |-- misc
|   |   |   |-- farro
|   |   |-- spelt
|   |-- grits
|   |-- western
|       |-- oats
|-- lemonade
|-- twizzlers
|-- veggies
    |-- celery
    |-- potato
        |-- yukon
        |-- russet
```

Notice how for `celery`, `oats`, and `potato` the vertical lines to the left of them do not extend all the way downward. The rule is this: all items in a directory are listed below their parent directory, indented by 4; reading left from the item name are a blank, two hyphens (--), and a pipe (|); above each pipe is another pipe, until we reach the parent directory name. Below each item in a directory are listed all its contents, or, if the item is either not a directory, or is a directory with no contents, nothing at all (unless there are further items from grandparent or other ancestor directories — see the "oats" line in the second example above).

A slightly less complex version of this is the following:

```
food
|-- dairy
|-- grains
|   |-- eastern
|   |   |-- quinoa
|   |   |-- misc
|   |   |   |-- farro
|   |   |-- spelt
|   |-- grits
|   |-- western
|   |   |-- oats
|-- lemonade
|-- twizzlers
|-- veggies
|   |-- celery
|   |-- potato
|   |   |-- yukon
|   |   |-- russet
```

where the "pipes" continue downward even if there's nothing left to connect to horizontally. If you manage to produce this less complex listing, you'll get partial but not full credit for this problem.

To create either of these versions of the output in OCaml, we'll represent the filesystem using

```
type file = Directory of (string * (file list)) | Plain of string ;;
```

Thus the "potato" directory above would be represented by

```
let pdir = Directory ("potato", [Plain "yukon"; Plain "russet"]) ;;
```

We'll represent the output of the waterfall procedure as a list of lists of characters. For our purposes, a "character" in OCaml is a single letter or symbol, enclosed in single quote-marks, like `'a'` or `'-'`, with one exception: a "newline" character is written `'\n'`. We've provided (copying from old OCaml source) a procedure `explode` that converts a string to a list of characters, and a helper procedure `pfall` to convert the output of `waterfall` to a printable string:

```
let rec explode (s: string): char list =
  let len = (String.length s)
  in match len with
     | 0 -> []
     | n -> s.[0] :: (explode (String.sub s 1 (len - 1))) ;;

let rec pfall (charmat: char list list) : string =
    String.concat "\n"
    (List.map (fun (alos: string list) -> List.fold_left (^) "" alos)
        (List.map (fun (aloc: char list) -> List.map Char.escaped aloc)
           charmat)) ;;
```

To visually test your procedure, you'll want to write something like

```
print_endline (pfall (waterfall my_dir));;
```

Your job is to write the `waterfall` procedure, which consumes a file, and produces a list of lists of characters representing the kind of display shown above. If the input is a plain file, your output should be the char list list corresponding to the filename. A directory, like the potato directory above, would have this output:

```
# waterfall pdir ;;
- =
[['p'; 'o'; 't'; 'a'; 't'; 'o'];
 ['|'; '-'; '-'; ' '; 'y'; 'u'; 'k'; 'o'; 'n'];
 ['|'; '-'; '-'; ' '; 'r'; 'u'; 's'; 's'; 'e'; 't']] : char list list
```

where we've indented to align things nicely.

```
#print_endline (pfall (waterfall pdir)) ;;
veggies
|-- celery
|-- potato
    |-- yukon
    |-- russet
```

Task: (3 points) Write a brief (2-4 sentence) explanation of your code's approach to the problem, guiding the person who has to grade it through the logic you use. Include this explanation as comments in your code submission file. If you need more than 4 sentences, that's fine, but you don't need to explain every detail– your design recipe comments should help with that. Tell us if you're doing something unusual ("I decided to first figure out where all the hyphens appear in the output, and only after that decide on the words to put in there.") so that we won't be baffled.

Task: (22 points) Write the `waterfall` procedure.

This problem is not easy, but it's fairly natural for OCaml. Some students may find the less complex version easier to write. Again, if you produce that version, you'll get partial, but not complete, credit.

3 Rising Sun Lists (Racket, 15 points)

Write a Racket procedure `shadows` that consumes a list of natural numbers and produces the list of *shadowed numbers* in the input list, in their input order. A number n is *shadowed* if some number after it in the list is *at least* n. For instance, in the list `(1 0 5 3 3)`, the numbers `1 0 3` are shadowed (where the '3' is from the second-to-last item in the original list). In other words `(shadows (list 1 0 5 3 3)` evaluates to the list `(list 1 0 3)`. Here's an additional example of expected functionality:

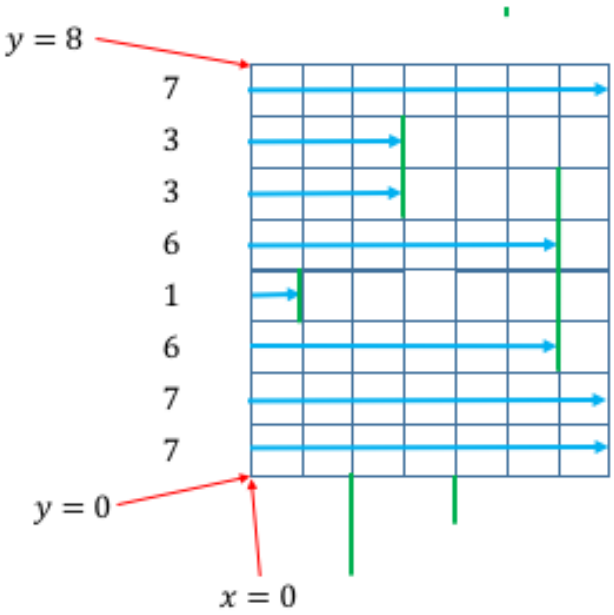
```
(shadows (list 1 0 0 0))
=> (list 0 0)
```

A linear-time procedure gets full credit; slower procedures that work correctly will get partial credit.

For those who are curious, the reason for this peculiar terminology comes from a theorem in mathematics, *which will be of no use to you in doing this problem*: The "Rising Sun Lemma" (check out the Wikipedia page) shows that for certain functions, some points of the graph of the function are "in shadow" when illuminated by the rising sun, far to the right (and then says profound things about how many such shadow-sets there might be, and of what kinds). Our definition above is the analogous one for natural-number lists.

4 Depth-casting (OCaml, 20 points)

Here's a picture of a problem that's at the core of some graphics algorithms. The picture shows a 2-dimensional version of what's a 3D problem in actual graphics.



We've got (on the right hand side) a collection of stuff, and we're making a "depth map" of it, a list of values that tells us how deep into the world we can see as we look along a horizontal ray at each of many possible heights.

We'll call the resulting depths "pixels", so each pixel (shown as a column of numbers at the left) records a "depth."

The image produced by this system is 1-dimensional: just a column of pixels, indicated by the numbers at the left. We say that the pixels form the "film plane" (although "film line" might be a better term), which is at $x = 0$. We've drawn the pixel values with a space to the right to separate them from the scene contents a little.

In this setup, a "camera" looks horizontally through each pixel from the left (the bright blue arrows), and simply records how far away is the first object it sees. (We call this *depth-casting*.)

The "objects" in the world are all vertical line segments (drawn in thick green), all at integer distances from the film plane at the far left.

If the ray for some pixel intersects no objects, then it hits a "back wall" whose distance is one greater than the distance of the farthest object.

The numbers at the left are the distance from the "film" (the line $x = 0$) to the objects or back wall within each row. The top ray and bottom two rays happen to hit the back wall.

We'll use x to denote the horizontal position of things in this picture, and y for the vertical positions. The bottom left of the picture is at $(x, y) = (0, 0)$; the first ray is at $y = 0.5$; the second ray at $y = 1.5$, and so on up to the topmost ray, at height $y = 7.5$.

A *scene* like the one shown here is specified by a *height* (the number of rows in the output, 8 in this example) and a sequence of *spans* indicating the positions of objects in each column. These follow some rules:

- There's an object in each column to the right of the film plane; these objects are at positions $x = 1, x = 2, x = 3, \ldots$. Each object is described by a pair (a, b) of integer heights, with $a \leq b$ This represents a span from height a to height b. Note that a or b may be less than zero or greater than the height; the only requirement is that $a \leq b$.
- The "back wall" is not represented by any object at all.
- The ray for pixel i shoots out of the center of that pixel, and will hit any object whose span has a nonzero overlap with the span $(i, i + 1)$.
- The spans in a scene are presented in left-to-right order, so the first span is at position $x = 1$, the second at position $x = 2$, and so on.

To indicate that a particular column has no visible span, we can simply use a span of size 0, like $(2, 2)$, which none of our rays can intersect.

A representation of the scene above is

```
height = 8
objects = ( (3 4) (-2 0) (5 7) (-1 0) (8 8) (2 6))
```

From the point of view of the picture produced, the span `(8 8)` could equally well have been `(9 11)` or `(-3,-3)`.

From the number of objects, you can infer the depth of the scene: because there are 6 spans in this scene, the back wall is at depth 7.

Task: (19 points) Write a procedure `depths` that consumes a scene and produces a depthmap image, i.e., a sequence of pixel depth-values, read from bottom to top. If the scene height is zero, the output depthmap should be empty.

In the example above, the output should be `[7;7;6;1;6;3;3;7]`, for instance. (Remember, we read them from bottom to top.)

The file `rays.ml` has the following type definitions:

```
type span = int * int      (* first must be no greater than second *)
```

```
type scene = int * (span list)  (* the first int is the scene height *)
type depthmap = int list  (* indicating how deep a ray from each point goes
   into the scene *)
```

And you are to fill in the definition written (in OCaml) as

```
(* header goes here *)
let rec depths ( (height, spans) : scene) : depthmap = ...
```

You should aim for a procedure that runs in big-O time proportional to $H * D$, where H is the height of the scene and D is the depth of the back wall. If you can do better than this, that's great, but you need not try to do so.

Hint: Although we've said that the rays are at heights $0.5, 1.5, \ldots$, there should be no need to actually use `float` values at all. The bottom-most blue ray, for instance, intersects a span `(a b)` only if $a <= 0$ and $b > 0$.

Task: (1 point) Describe in a few sentences your general approach to the problem, making it easy for the grader to understand your code. Include these sentences as comments in your code submission file.

5 Analysis (20 points)

Normally an analysis problem would start from code for some procedure, ask you to derive a recurrence, and then ask you to solve it and provide a big-O bound on the runtime of the procedure. A student who gets the recurrence wrong has a hard time with the remaining parts, and grading that stuff is horrible. So we've broken this into two pieces: one where you derive a recurrence, and the other where you analyze a different and *unrelated* recurrence.

We've provided code, `flop.ml`, to produce a *flop tree* from a list of integers. A flop tree's root contains the int list, its left child is a flop-tree made from the reverse of the first third of the list, and its right child is a flop-tree made from the reverse of the last third of the list. So a flop-tree for the list of digits from 1 to 9 contains that list at its root; the left child is a tree with root [3; 2; 1], and the right child has root [9; 8; 7]. The left child's left child contains 3 and a pair of `Leafs`; its right child contains 1 and a pair of `Leafs`. The right child that had [9; 8; 7] at the root has a left child (9, Leaf, Leaf) and a right child (7, Leaf, Leaf). When the number of nodes isn't divisible by 3, we round down. You can look at and experiment with the code to determine what it's doing.

Let $F(n)$ be the number of elementary operations performed by `flop` on an input list of length n. (We don't say "the greatest number" because the amount of work is determined by the length of the list rather than its contents: any 12-element list takes the same amount of work).

Task: Write a recurrence relation for the function $F : \mathbf{N} \to \mathbf{N}$, and explain each line of the recurrence in a sentence or two (e.g., "sorting in line 3 of the code takes time $Cn \log n$, and the append takes time proportional to $2n$").

Task: Now suppose that $H : \mathbf{N} \to N$ is a function satisfying the recurrence

$$H(0) = A \tag{1}$$
$$H(1) = A \tag{2}$$
$$H(n) \leq 3n + H(\text{floor}(n/2)) \qquad \text{for } n > 1 \tag{3}$$

Use plug-n-chug to conjecture a bound on H, prove your bound correct via a well-ordering proof, and then conclude something about the big-O category of H.

Hint: You may find it useful to limit your plugging and chugging to those cases in which the input n is a power of two.

Put your answers to both questions in the provided proof template in `analysis.tex`; we'll run pdflatex to produce the PDF that gets graded.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/cs017/feedback`.