CS51 PROBLEM SET 6:
REFS, STREAMS, AND MUSIC

STUART M. SHIEBER

Introduction

In this problem set you will work with two new ideas: First, we provide a bit of practice with imperative programming emphasizing mutable data structures and the interaction between assignment and lexical scoping. Since this style of programming is probably most familar to you, this portion of the problem set is brief. Second, we introduce lazy programming and its use in modeling infinite data structures. This part of the problem set is more extensive, and culminates in a project to generate infinite streams of music.

As in the previous assignments, all of your programs must compile in order to receive credit. **Programs that do not compile will receive no credit.** Make sure that the functions you are asked to write have the correct names and the number and type of arguments specified in the assignment. Finally, please pay attention to good design and style and follow the style guidelines posted on the course web site. Think carefully before writing the code, and try to come up with simple, elegant solutions. As usual, to download the problem set, follow the instructions found here.

- **Testing:** This week, we are only requiring that you write explicit tests for Section 1, placed in the file `refs_test.ml`. For each of the functions that you write in this part, you should provide tests covering as broad a range of functionality of the functions as possible so as to convince yourself that they work. For the other sections, you are not required to submit tests, but you ought to convince yourself that your solution is correct. For Section 3, testing using traditional methods is difficult but there is an easy (and hopefully fun) way to determine if your code works.
- **Grading:** Your code will be evaluated on correctness, design (including testing), and style (as discussed in the style guide).
- **Collaboration policy:** You are encouraged to work with a partner on this problem set. The course collaboration policy still applies to the pairs.
- **Time Spent:** At the bottom of each file, you will notice a prompt for you to enter the approximate number of minutes that you spent (per person, averaged) working on that part. We are interested in your sincere answers, and this will help us tailor future problem sets.

Date: March 28, 2018.

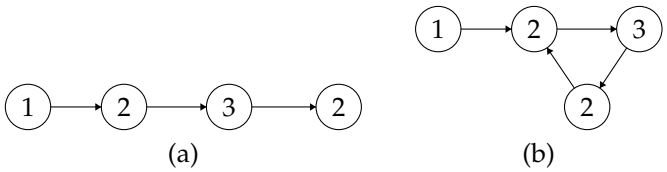

FIGURE 1. Example mutable lists: (a) no cycle; (b) with cycle.

1. MUTABLE LISTS AND CYCLES

We've provided some problems involving mutable lists in `refs.ml`. *Remember that you must test each function in this part in* `refs_test.ml`. Your solutions will not be graded for time or space efficiency.

Recall from lecture that a major difficulty with mutable lists is that it is possible to introduce cycles, links to elements that appeared earlier in the list, which cause a naive traversal of the list to loop forever.

Figure 1(a) shows an example of a mutable list that doesn't have a cycle, even though the same element is repeated twice, whereas Figure 1(b) is an example of a mutable list with a cycle.

Problem 1. *Write a function* `has_cycle` *that returns* `true` *if and only if a mutable list has a cycle. Your function must not alter the original mutable list, and it must terminate eventually.*

Think about how you will know if a node that you are visiting has been seen before. Testing whether the reference `a_ref` *points to* `b` *can be done with* `(!a_ref) == b`*. The* `(==)` *function tests equality at the level of memory location rather than value.* □

Problem 2. *Write a function* `flatten` *that takes a mutable list and removes any cycle in it destructively by removing backward links. This means that the data structure should be changed* in-place*, such that the list passed as an argument itself is altered if necessary. Note that this is very different from the functional programming approach that we have been using up to this point, where functions might return an altered* copy *of a data structure. Suppose you pass in the mutable list from Figure 1(b);* `flatten` *should alter it such that it looks like Figure 1(a), and then return* `unit`*. If you are unsure how to destructively alter a mutable list, take a look at* `reflist` *in* `refs_test.ml`*.* □

Problem 3. *Write* `mlength`*, which finds the number of elements in a mutable list. This should always terminate,* even if the list has a cycle*. For example, both Figures 1(a) and (b) have length 4. The* `mlength` *function must be* nondestructive*, that is, the original mutable list should not change.* □

Problem 4. ***Challenge problem:*** *It's possible to complete Problem 1 in such a way that it doesn't use any additional space other than that taken up by the list passed as an argument. Attempt this only if you've finished the rest of the assignment and are up for a challenge.* □

2. LAZY EVALUATION

In this section you'll gain practice with lazy evaluation using the OCaml `Lazy` module through problems with infinite streams and infinite trees.

2.1. **Series acceleration with infinite streams.** In lecture, we showed how to use Taylor series to approximate π. The code needed to do so is provided in a module `NativeLazyStreams`. The method relies on generating the series of terms of a Taylor series, computing the partial sums that approximate π, and then finding a pair of consecutive approximations that are within the desired tolerance ϵ, as shown here:

```
# let pi_sums = sums pi_stream ;;
val pi_sums : float stream = <lazy>
# first 5 pi_sums ;;
- : float list =
[4.; 2.6666666666666696; 3.4666666666666679; 2.89523809523809561;
 3.33968253968254025]
# within 0.1 pi_sums ;;
- : int * float = (19, 3.09162380666784)
# within 0.01 pi_sums ;;
- : int * float = (199, 3.13659268483881615)
# within 0.001 pi_sums ;;
- : int * float = (1999, 3.14109265362104129)
```

The method works, but converges quite slowly. Notice that it takes some 200 terms in the expansion to get within 0.01 of π. In this section of the problem set, you will use a technique called *series acceleration* to speed up the process of converging on a value. A simple method is to average adjacent elements in the approximation stream. The necessary code for you to use and modify can be found in the file `streamstrees.ml`.

Problem 5. *Write a function* `average` *that takes a* `float stream` *and returns a stream of floats each of which is the average of adjacent values in the input stream. For example:*

```
# first 5 (average (to_float nats)) ;;
- : float list = [0.5; 1.5; 2.5; 3.5; 4.5]
```

You should then be able to define a stream `pi_avgs` *of the averages of the partial sums in* `pi_sums`*. How many steps does it take to get within 0.01 of* π *using* `pi_avgs` *instead of* `pi_sums`*?* □

An even better accelerator is Aitken's method. Given a sequence s, Aitken's method generates the sequence s' given by

$$s'_n = s_n - \frac{(s_n - s_{n-1})^2}{s_n - 2s_{n-1} + s_{n-2}}$$

Problem 6. *Implement a function* `aitken` *that applies Aitken's method to a* `float stream` *returning the resulting* `float stream`*.* □

Problem 7. *Try the various methods to compute approximations of* π *and fill out the table in the* `streamstrees.ml` *file with what you find.* □

2.2. **Infinite trees.** Just as streams are a lazy form of list, we can have a lazy form of trees. In the definition below, each node in a lazy tree of type `'a tree` holds a value of some type `'a`, and a (conventional, finite) list of one or more (lazy) child trees.

```
type 'a treeval = Node of 'a * 'a tree list
 and 'a tree = 'a treeval Lazy.t ;;
```

Problem 8. *Complete the implementation by writing functions* `node`, `children`, `print_depth`, `tmap`, `tmap2`, `bfenumerate`, `onest`, `levels`, *and* `tree_nats` *as described in* `streamstrees.ml`. *We recommend implementing them in that order.* □

3. The song that never ends

In this part, you will explore a fun application of streams: music. Before you begin, if you're not already familiar with basic music notation and theory, you should learn some of these concepts. Numerous online tutorials exist, such as this one, though if you're short on time, the following introduction should be sufficient for this assignment.

3.1. **A brief introduction to music theory.** The major musical objects we use in this assignment are *notes* and *rests*. Notes indicate when a sound is heard, and rests indicate a certain period of silence. Notes have a pitch and a duration, rests have only a duration. A pitch has one of twelve names: C, D♭ (pronounced D-flat, also called C♯, pronounced C-sharp), D, E♭ (also called D♯), E, F, G♭ (F♯), G, A♭ (G♯), A, B♭ (A♯), B. The distance between two consecutive pitches in this list is called a *half-step*. Each of these pitches can be played in many *octaves*. For example, a piano keyboard has many Cs. The one in the middle of the keyboard is called "middle C", but the key 12 half-steps (that is, twelve piano keys counting white and black keys) above and below it are both Cs as well, in different octaves. Thus, a pitch can be thought of as a name and an octave.

A basic unit of time in music is called the *measure*, and notes are named based on what fraction of a measure they last. The most common notes are *whole notes*, *half notes*, *quarter notes* and *eighth notes.* Rests can be similarly named for their duration, and so there are *whole rests*, *half rests* and so on.

3.2. **Data types for musical objects.** The top of `music.ml` provides simple definitions for musical data types. The type `pitch` is a tuple of a `p`, which has one constructor for each of the 12 pitch names, and an integer representing the octave. (Note that our definition includes only flats and not sharps. We apologize if this offends any music theorists.) There is also a data type representing musical objects, `obj`. An `obj` can either be a `Note`, which contains a pitch, the duration as a float, and the volume (how loud the note is played) as an integer from 0 to 128; or a `Rest`, which contains the duration as a float. Durations are in units of measures, so 1.0 is a whole note or rest, 0.5 is a half note or rest, and so on. You may also indicate other float values, like 0.75 (called a *dotted half* note or rest) or 1.5, 2.0, 3.0, and so forth, for multi-measure notes and rests.

This data type is useful for representing the kind of music notation you'd see in sheet music, but what does this have to do with streams? For the purpose of representing or playing music on a computer (for example, in a MIDI sequencer), a different representation is more convenient. In this representation, a piece of music is a stream of events, where an event is the start of a certain tone or the end of a certain tone. Both kinds of events carry two pieces of information: how much time should elapse between the previous event and this one, and the pitch of the event. Note that stops, in addition to starts, must have pitch. This is so that we can have multiple notes played at a time.

The sequencer has to know which of the notes currently playing we want to stop. Starts also have a volume. (It so happens that in the MIDI specification a stop can be represented like a start with a volume of 0.) A data type for musical events is also given at the top of `music.ml`. An `event` can be a `Tone` of a time, a pitch, and a volume, or a `Stop` of a time and a pitch. *Remember that the times stored with events are relative to the previous event and not absolute.*

A natural way to represent music is as a stream of events. For this purpose, you'll use the `NativeLazyStreams` module. The sequencer doesn't care about the rest of the piece, it just takes events as they come and processes them. This also allows us to lazily perform actions on an entire piece of music. The downside of this is that, because our streams are always infinite, songs represented in this way can't end. With the exception of certain songs commonly performed in the back seats of cars on family trips, pieces of music are of finite length, but we'll get back to this later.

Problem 9. *Write a function* `list_to_stream` *that builds a music stream (*`event stream`*) from a finite list of musical objects. The stream should repeat this music forever. Also look out for invalid lists which don't represent any meaningful music.*

Hint: *Use a recursive helper function as defined, which will change the list but keep the original list around as* `lst`*. Both need to be recursive, since you will call both the inner and outer functions at some point. If you'd rather solve the problem a different way, you may remove the definition of the recursive helper as long as you keep the type signature of* `list_to_stream` *the same.* □

Problem 10. *Write a function* `pair` *that merges two event streams into one. Events that happen earlier in time should appear earlier in the merged stream. Events in the merged stream should happen at the same absolute time they happened in the individual streams. For example, if one stream has a start at time 0, and a stop at time 0.5 after that, and the other stream has a start at time 0.25 and a stop at time 0.5 after that, the combined stream should have a start at time 0, the second start at 0.25 after that, the first stop at 0.25 after that, and the second stop at 0.25 after that. This will require some careful thought to update the time differences of the event that is not chosen.* □

Problem 11. *Write a function* `transpose` *that takes an event stream and transposes it (moves each pitch up by a given number of* `half_steps`*.) For example, a C transposed up 7 half-steps is a G. A C transposed up 12 half-steps is the C in the next octave up. Use the helper function* `transpose_pitch`*, which transposes a single pitch value by the correct number of half-steps.* □

What fun would an assignment about music be if there was no way to hear music? We've given you a function `output_midi` that takes a string `filename`, integer `n`, and an event stream `str` and outputs the first `n` events of `str` as a MIDI file with the given filename. Note that the integer argument is necessary because streams are infinite and we can't have infinite MIDI files. To hear some music and test the functions you've written so far, uncomment the lines of code under the comment "Start off with some scales." When you compile and run the file at the command line or execute the file's definitions in the OCaml toplevel, this will output a file `scale.mid`, which plays a C major scale together with a G major scale. You should be able to play the generated MIDI file with an application such as GarageBand on Mac OS or Windows Media Player on Windows.

Please watch this informative YouTube video that relates to the next problem.

As any fourth-grader in school band can tell you, scales are boring. This example also doesn't show us why we should care about the power music streams give us. For a more realistic example, we need a piece of music that is repetitive, and has several parts played together which resemble each other with small modifications. To be a good example, it should also be instantly recognizable and in the public domain so we don't get sued. Such a piece is Johann Pachelbel's *Canon in D major*. (We guarantee you've heard it.)

We'll be building a music stream of a simplified version of eight measures of Pachelbel's canon. This segment contains a *basso continuo*, defined as the event stream `bass`. This is a two-measure pattern that repeats over and over (ah, so this is where infinite streams are useful.) It also contains a melody, of which six measures are contained in the stream `melody`. The bass plays for two measures, and then the melody starts and *plays over it*. Two measures later, the melody *starts again*, playing over the bass and the first melody. Finally, two measures after this (six measures from the start), the melody starts once more. For the final two measures, there are four streams playing at once: the bass and three copies of the melody, at different points.

Problem 12. *Define a stream* `canon` *that represents this piece. Use the functions* `shift_start` *and* `pair`*, and the streams* `bass` *and* `melody`*. When you're done, uncomment the line below the definition and compile and run the code. It will produce a file canon.mid which, if you've done the problem correctly, will contain the first eight measures of the piece. (We export 176 events since these eight measures should contain 88 notes; you could increase this number if you want to hear more music, though it won't be true to the original.) If it sounds wrong, it probably is. If it sounds right, either it is or you're a brilliant composer. (Note: As this is not a music class, brilliant composers will not earn extra points on this question.)* □

Problem 13. ***Challenge problem:*** *There's lots of opportunity for extending your system. If you want, try implementing other functions for manipulating event streams. Maybe one that increases or decreases the timescale, or one that doubles every note, or increases or decreases the volume. See what kind of music you can create. We've given you some more interesting streams of music to play around with if you'd like, named* `part1` *through* `part4`*. They're all the same length, 2.25 measures.* □

4. Submission

Before submitting, please estimate how much time you and your partner each spent on each section of the problem set by editing the line in each file that looks like

```
let minutes_spent_on_part () : int = failwith "not provided" ;;
```

to replace the value of the function with an approximate estimate of how long (in minutes) the part took you to complete. Make sure it still compiles.

Then, to submit the problem set, follow the instructions found here. Please note that only one of your partners needs to submit the problem set.