CS-3MI3 Homework 2 Sample Code

Mark Armstrong, PhD Candidate, McMaster University

September 20th, 2019

This code is provided to help with concepts needed in homework 2.

Learning from the literate portions of this file

The literate portions of this document contain explanations about *how* to define/work with new types (or an existing type, in the case of hashes) in Ruby and F#.

You may take this document as a guide for your literate writeups, but usually, in your documents, you should instead discuss implementation details (what and why) instead of explaining constructs (how).

Some implementation details are discussed here in comments on the code. So in your documents, you may wish to emulate

- the form of this document, by
- the content of the code comments.

A custom type of lists

Homework 2 asks you to build a type of trees using

- a class in Ruby, and
- an algebraic datatype in F#.

A similar, but slightly simpler task, would be to build our own type of lists in each language.

A custom type of lists in Ruby

Being primarily an object-oriented language, in Ruby, we implement our own types as *classes*.

In this case, I will construct a class representing lists. These lists are inductive; every list is either empty, or an element (named `here`) followed by a list (named `next`).

Of course, since Ruby is dynamically typed, it is up to me as the implementor to ensure that the types of `here` and `next` are correct at all times.

To define a class, start with the keyword `class` followed by a name for your new class.

```
# My own class of lists.
# Note that these lists misbehave if you try to store nil in them;
# specifically, lists which start with nil are treated as empty,
# so everything following a nil in the list will be ignored.
class MyList
```

Ruby is a heavily dynamic language, and so usually the fields (a.k.a., attributes, data, member variables) of a class do not need to be declared. However, it can be beneficial to declare them using one of the keywords `attr_reader`, `attr_writer` or `attr_accessor`, which create read, write and read/write methods automatically. In the case of this custom list types, I want fields to be read-only, so I make only readers for them.

```
  # Create readers for two fields, here and next.
  attr_reader :here
  attr_reader :next
```

If we want to perform some initialisation when creating an object of this class (via the `MyList.new` method which is inherited from the implicit super-class `Class` of `MyList`), you can define an `initialize` method to do so. Note here that the local fields `next` and `here` are referred to as `@here` and `@next` inside the class. (Static fields, whose value would be shared by all objects, would have their names prepended by `@@`).

```
  # Initialise a list as a singleton or an empty list.
  def initialize(val=nil)
```

```
    @here = val
    @next = nil
    return self
  end
```

Our `initialize` method allows us to initiate lists as being empty or singleton. We now need some method to build up lists; here, I define `prepend` to do so.

```
  # Add a value to the beginning of a list
  # Note that we test if the head (@here) is tested against nil;
  # lists with a nil head are considered to be empty.
  def prepend(val)
    if @here != nil
      # Reproduce the current list, making it the new next list
      @next = self.clone()
    end
    @here = val
    return self
  end
```

For convenience, we also provide a method to catenate lists. We clone the list provided as argument and make it the end of this list.

```
  def catenate(l)
    if @here != nil and @next != nil
      # keep recursing until we reach the end of our list
      @next = @next.catenate(l)
    elsif @here != nil
      # at the end, attach a copy of l
      @next = l.clone()
    else
      # there is no list here; just copy l
      @here = l.here
      @next = l.next
    end
    return self
  end
```

We want our lists to be printable, so we override the `to_s` method to print their contents, rather than just printing out an identifier for the object.

```
  def to_s
    if @here and @next
      return @here.to_s + " :: " + @next.to_s
    elsif @here
      return @here.to_s + " :: eol"
    else
      return "eol"
    end
  end
```

Like any unit of code in Ruby, class declarations end with `end`.

```
end
```

Now we can build a list and print it using the methods defined above.

```
# The list containing 1, prepended with 2, catenated with the list containing 3
# 2  1  3  eol
puts ((MyList.new(1)).prepend(2)).catenate(MyList.new(3))
```

A custom type of lists in F#

In F#, we have *algebraic* or *inductive* datatypes as the principal method for defining new types.

Since lists *are* an inductive datatype, representing them in this way is trivial.

A list is either empty or the catenation of an integer with a list.

```
type MyList = Empty | Cons of int * MyList
```

Then we can write lists as follows.

```
let singleton = Cons(1, Empty) // 1  []
let binary = Cons(2, singleton) // 2  1  []
let ternary = Cons(1,Cons(2,Cons(3,Empty))) // 1  2  3  []
```

A nice advantage of algebraic datatypes is that they can be pretty-printed without any additional effort; the output is simply the sequence of constructors that make it up.

```
printfn "A singleton list: %O" singleton
printfn "A two-element list: %O" binary
printfn "A three-element list: %O" ternary
```

The hash type in Ruby

Homework 2 also asks you to convert your tree type to and from hashes.

Hashes, also called associative arrays, are another type of collection available in many languages.

```
list_like_hash = { 0 => "this", 1 => "that", 2 => "the other" }

puts list_like_hash
puts "#{list_like_hash[0]}, #{list_like_hash[1]} and the #{list_like_hash[2]}"
```

Hashes in Ruby can be heterogeneous, storing elements of different types.

```
heterogeneous_hash = {0 => 0, 1 => "uh... one?"}

puts heterogeneous_hash
```

Note that lists are also heterogeneous in Ruby!

```
heterogeneous_list = [0, "uh... one?"]

puts heterogeneous_list
```

The difference between hashes and lists are that lists are indexed by numbers, starting from 0. Hashes can be indexed by any type (and the types of keys can vary for a single hash, too!). Note that hashes also have the `each` method for looping over the contents.

```
owing = {"Bob" => 200, "Larry" => -3, "Archibald" => 9001}

owing.each{ |p,a| puts "#{p} owes me #{a}" }
```