HW 5 – Algebraic Datatypes

CS 421 – Fall 2015
Revision 1.0

Assigned September 22, 2015
Due September 30, 2015, 23:59 pm
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objective

The purpose of this HW is to test your understanding of the algebraic datatype system in OCaml.

3 Turn-In Procedure

HW 5 requires you to write OCaml code, so before submitting please make sure that your solution successfully compiles. You should put code answering each of the problems below in a file called `hw5.ml`.

The command to commit this file is:

```
svn commit -m "Turning in hw5." hw5.ml
```

4 Background

4.1 Algebraic Data Types

Algebraic datatypes in OCaml allow us to do is create containers in which we can store other values or containers. We can also create different variations on those containers and have them all create objects of the same type.

```
type owner = Name of string;;

type vehicle = Hovercar of owner
             | Spaceship of (owner * string)
             | Submarine of (owner * int)
             | Bucket of (owner * float);;
```

In this example, despite `Hovercar`, `Spaceship`, and `Submarine` being different "containers" (or *constructors*) that hold different types, they all produce objects of the same type: namely, `vehicle`.

```
# let planet_express_ship =
  Spaceship (Name "Hubert J. Farnsworth", "Planet Express Ship");;
val planet_express_ship : vehicle =
  Spaceship (Name "Hubert J. Farnsworth", "Planet Express Ship")
# let washbucket = Bucket (Name "Scruffy", 1.414213);;
val washbucket : vehicle = Bucket (Name "Scruffy", 1.414213)
```

If we construct expressions (like we have with `planet_express_ship` and `washbucket`), not only do we get to package up the data we want to keep with a specific kind of object, like a `Spaceship` or a `Bucket`, but they also make objects of the same type - in this case, `vehicle`.

4.2 Representing a Simplified Markup Language

Markup languages allow text to be composed of regions, some of which are plain text and some that are labeled (or *marked up*) with additional information. Further, the regions may contain arbitrarily deep nestings of sub-regions with additional markups. Consider an example in an XML-like language with content which looks like the following:

```
<employee>
  <name>Philip J. Fry</name>
  <job>Delivery Boy</job>
  <supervisor>
    <employee>
      <name>Hubert J. Farnsworth</name>
      <job>CEO</job>
    </employee>
  </supervisor>
</employee>
```

Within each matched pair of labels, or tags (e.g. `<employee> ...</employee>`) are either additional tags or a string containing the data as indicated by the tag. Also notice that within each tag is a string (in this example, `employee`) which denotes the type of tag it is.
Now, consider a JSON-like representation of the same example:

```
{
  employee: {
    name: "Philip J. Fry",
    job: "Delivery Boy",
    supervisor: {
      employee: {
        name: "Hubert J. Farnsworth",
        job: "CEO",
      }
    }
  }
}
```

In both cases, we're able to capture the same information despite clear differences in the concrete syntax. More concretely, even though the XML-like example and the JSON-like example use different syntax, they both represent the same data.

Your job is to come up with an algebraic datatype that models the data in an expression in a simplified markup language that could be presented in XML or JOSn. That is, we want an algebraic datatype that allows us to model objects that have a label with a name (e.g. `employee`) has a value, where the value is either a string of text (e.g. `Philip J. Fry`), or a list of objects.

5 Problems

1. (10 pts) Define a datatype `markup` that models objects, where an object has a string label and a value that is either a string or a list of further objects. Your datatype should model exactly all and only objects as described above. Your datatype should be able to represent the example given in XML and JSON in the previous section. If need be, you are allowed to create other datatypes in addition to `markup`.

2. (3 pts) Using your datatype `markup`, define a variable `crew` that models the data given in the `employee` examples of the previous section.

```
let crew = ... ;;
val crew : markup = ...
```

3. (10 pts) Write a function `countNameInstances : markup -> string -> int` that takes some markup `m` and a string `s`, and counts the number of times `s` appears as a tag in `m`.

```
let countNameInstances s = ... ;;
- : markup -> string -> int = <fun>

countNameInstances crew "job";;
- : int = 2
```