Exploring Languages with Interpreters and Functional Programming Chapter 21

H. Conrad Cunningham

11 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

<

21 Algebraic Data Types

21.1 Chapter Introduction

The previous chapters have primarily used Haskell's primitive types along with tuples, lists, and functions.

The goals of this chapter (21) are to:

- describe how to define and use of Haskell's (user-defined) algebraic data types
- introduce improvements in error-handling using `Maybe` and `Either` types
- present a few larger programming projects

Algebraic data types enable us to conveniently leverage the power of the type system to write safe programs. We extensively these in the remainder of this textbook.

The Haskell source module for this chapter is in file `AlgDataTypes.hs`.

TODO: It might be better to factor the source code into multiple files.

21.2 Concepts

An *algebraic data type* [4,27,30] is a type formed by combining other types, that is, it is a *composite* data type. The data type is created by an algebra of operations of two primary kinds:

- a *sum* operation that constructs values to have one variant among several possible variants. These sum types are also called *tagged*, *disjoint union*, or *variant* types.

  The combining operation is the alternation operator, which denotes the choice of one but not both between two alternatives.

- a *product* operation that combines several values (i.e., *fields*) together to construct a single value. These are *tuple* and *record* types.

  The combining operation is the Cartesian product from set theory.

We can combine sums and products recursively into arbitrarily large structures.

An *enumerated type* is a sum type in which the constructors take no arguments. Each constructor corresponds to a single value.

Aside: ADT confusion

Although sometimes the acronym ADT is used for both, an *algebraic data type* is a different concept from an *abstract data type* [14,29].

- We specify an algebraic data type with its *syntax* (i.e., structure)—with rules on how to compose and decompose them.

- We specify an abstract data type with its *semantics* (i.e., meaning)—with rules about how the operations behave in relation to one another.

  The modules we build with abstract interfaces, contracts, and data abstraction, such as the Rational Arithmetic modules from Chapter 7, are abstract data types.

Perhaps to add to the confusion, in functional programming we sometimes use an algebraic data type to help define an abstract data type. We do this in the Carrie's Candy Bowl project at the end of this chapter. We consider these techniques more fully in Chapter 22.

21.3 Haskell Algebraic Data Types

21.3.1 Declaring data types

In addition to the built-in data types we have discussed, Haskell also allows the definition of new data types using declarations of the form:

`data` **Datatype a*1* a*2*** $\cdots$ **a*n*** = **Cnstr*1*** | **Cnstr*2*** | $\cdots$ | **Cnstr*m***

where:

- **Datatype** is the name of a new type constructor of *arity* ***n*** *(n ≥ 0)*. As with the built-in types, the name of the data type must begin with an uppercase letter.
- **a*1***, **a*2***, $\cdots$ **a*n*** are distinct type variables representing the ***n*** parameters of the data type. These begin with lowercase letters (by convention at the beginning of the alphabet).
- **Cnstr*1***, **Cnstr*2***, $\cdots$, **Cnstr*m*** are the ***m*** ($m \geq$ 1$) data constructors that describe the ways in which the elements of the new data type are constructed. These begin with uppercase letters.

The `data` definition can also end with an optional `deriving` that we discuss below.

21.3.2 Declaring type `Color`

For example, consider a new data type `Color` whose possible values are the colors on the flag of the USA. The names of the data constructors (the color constants in this case) must also begin with capital letters.

```
data Color = Red | White | Blue
             deriving (Show, Eq)
```

`Color` is an example of an *enumerated type*, a *sum* type that consists of a finite sequence of *nullary* (i.e., the arity—number of parameters—is zero) data constructors.

We can use the type and data constructor names defined with `data` in declarations, patterns, and expressions in the same way that the built-in types can be used.

```haskell
isRed :: Color -> Bool
isRed Red = True
isRed _   = False
```

Data constructors can also have associated values. For example, the constructor `Grayscale` below takes an integer value.

```haskell
data Color' = Red' | Blue' | Grayscale Int
              deriving (Show, Eq)
```

Constructor `Grayscale` implicitly defines a constructor function with the type.

21.3.3 Deriving class instances

The optional `deriving` clauses in the above definitions of `Color` and `Color'` are very useful. They declare that these new types are automatically added as instances of the type classes listed.

Note: Chapter 23 explores the concepts of type class, instance, and overloading in more depth.

In the above cases, `Show` and `Eq` enable objects of type `Color` to be converted to a `String` and compared for equality, respectively.

The Haskell compiler derives the body of an instance syntactically from the data type declaration. It can derive instances for classes `Eq`, `Ord`, `Enum`, `Bounded`, `Read`, and `Show`.

The derived instances of type class `Eq` include the `(==)` and `(/=)` methods.

Type class `Ord` extends `Eq`. In addition to `(==)` and `(/=)`, a derived instance of `Ord` also includes the `compare`, `(<)`, `(<=)`, `(>)`, `(>=)`, `max`, and `min` methods. The ordered comparison operators use the order of the constructors given in the `data` statement, from smallest to largest, left to right. These comparison operators are strict in both arguments.

Similarly, a derived `Enum` instance assigns successive integers to the constructors increasing from 0 at the left. In addition to this, a derived instance of `Bounded` assigns `minBound` to the leftmost and `maxBound` to the rightmost.

The derived `Show` instance enables the function `show` to convert the data type to a syntactically correct Haskell expression consisting of only the constructor names, parentheses, and spaces. Similarly, `Read` enables the function `read` to parse such a string into a value of the data type.

For example, the data type `Bool` might be defined as:

```haskell
data Bool = False | True
            deriving (Ord, Show)
```

Thus `False < True` evaluates to `True` and `False > True` evaluates to `False`. If `x == False`, then `show x` yields the string `False`.

21.3.4 Exploring more example types

Consider a data type `Point` that has a type parameter. The following defines a polymorphic type; both of the values associated with the constructor `Pt` must be of type `a`. Constructor `Pt` implicitly defines a constructor function of type `a -> a -> Point a`.

```
data Point a = Pt a a
               deriving (Show, Eq)
```

As another example, consider a polymorphic set data type that represents a set as a list of values as follows. Note that the name `Set` is used both as the type constructor and a data constructor. In general, we should not use a symbol in multiple ways. It is acceptable to double use only when the type has only one constructor.

```
data Set a = Set [a]
             deriving (Show, Eq)
```

Now we can write a function `makeSet` to transform a list into a `Set`. This function uses the function `nub` from the `Data.List` module to remove duplicates from a list.

```
makeSet :: Eq a => [a] -> Set a
makeSet xs = Set (nub xs)
```

As we have seen previously, programmers can also define type synonyms. As in user-defined types, synonyms may have parameters. For example, the following might define a matrix of some polymorphic type as a list of lists of that type.

```
type Matrix a = [[a]]
```

We can also use special types to encode error conditions. For example, suppose we want an integer division operation that returns an error message if there is an attempt to divide by 0 and returns the quotient otherwise. We can define and use a union type `Result` as follows:

```
data Result a = Ok a | Err String
                deriving (Show, Eq)

divide :: Int -> Int -> Result Int
divide _  0 = Err "Divide by zero"
divide x  y = Ok (x `div` y)
```

Then we can use this operation in the definition of another function `f` that returns the maximum `Int` value `maxBound` when a division by 0 occurs.

```
f :: Int -> Int -> Int
f x y  = return (divide x y)
         where return (Ok z)  = z
               return (Err s) = maxBound
```

The auxiliary function `return` can be avoided by using the Haskell `case` expression as follows:

```
f' x y =
    case divide x y of
        Ok z  -> z
        Err s -> maxBound
```

This case expression evaluates the expression `divide x y`, matches its result against the patterns of the alternatives, and returns the right-hand-side of the first matching patter.

Later in this chapter we discuss the `Maybe` and `Either` types, two polymorphic types for handling errors defined in the Prelude.

21.4 Recursive Data Types

Types can also be recursive.

21.4.1 Defining a binary tree type

For example, consider the user-defined type `BinTree`, which defines a binary tree with values of a polymorphic type.

```
data BinTree a = Empty | Node (BinTree a) a (BinTree a)
                 deriving (Show, Eq)
```

This data type represents a binary tree with a value in each node. The tree is either "empty" (denoted by `Empty`) or it is a "node" (denoted by `Node`) that consists of a value of type `a` and "left" and "right" subtrees. Each of the subtrees must themselves be objects of type `BinTree`.

Thus a binary tree is represented as a three-part "record" as shown in on the left side of Figure 21.1. The left and right subtrees are represented as nested binary trees. There are no explicit "pointers".

Consider a function `flatten` to return the list of values in binary tree in the order corresponding to a left-to-right in-order traversal. Thus expression

```
flatten (Node (Node Empty 3 Empty) 5
              (Node (Node Empty 7 Empty) 1 Empty))
```

yields `[3,5,7,1]`.

```
flatten :: BinTree a -> [a]
flatten Empty        = []
flatten (Node l v r) = flatten l ++ [v] ++ flatten r
```

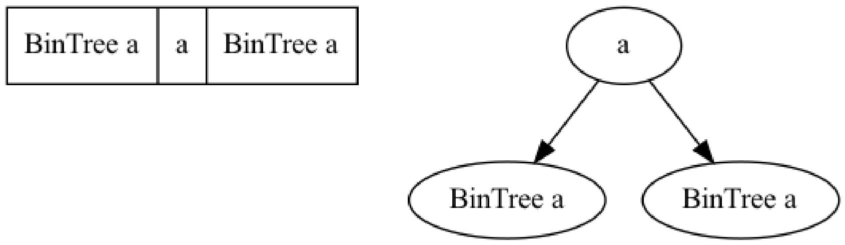

Figure 21.1: Binary tree `BinTree`.

The second leg of `flatten` requires two recursive calls. However, as long as the input tree is finite, each recursive call receives a tree that is simpler (3.g., shorter) than the input. Thus all recursions eventually terminate when `flatten` is called with an `Empty` tree.

Function `flatten` can be rendered more efficiently using an accumulating parameter and cons as in the following:

```
    flatten' :: BinTree a -> [a]
    flatten' t = inorder t []
                 where inorder Empty xs        = xs
                       inorder (Node l v r) xs =
                           inorder l (v : inorder r xs)
```

Auxiliary function `inorder` builds up the list of values from the right using cons.

To extend the example further, consider a function `treeFold` that folds an associative operation `op` with identity element `i` through a left-to-right in-order traversal of the tree.

```
    treeFold :: (a -> a -> a) -> a -> BinTree a -> a
    treeFold op i Empty        = i
    treeFold op i (Node l v r) = op (op (treeFold op i l) v)
                                    (treeFold op i r)
```

21.4.2 Exporting types from modules

If an algebraic data type is defined in a module, we can export the type and make it available to users of the module. Suppose the `BinTree` type and the functions above are defined in a Haskell module named `BinaryTrees`. Then the following module header would export the type `BinTree`, the three explicitly defined functions, and the functions generated for the `Eq` and `Show` classes.

```
    module BinaryTrees
        ( BinTree, flatten, flatten', treeFold )
```

```
where  -- implementation details of type and functions
```

This module definition makes the type `BinTree` and its two constructors `Node` and `Empty` available for use in a module that imports `BinaryTrees`.

If we want to make the type `BinTree` available but not its constructors, we can use the following module header:

```
module BinaryTrees
    ( BinTree(..), flatten, flatten', treeFold )
where  -- implementation details of type and functions
```

With `BinTree(..)` in the export list, `BinTree` values can only be constructed and examined by functions defined in the module (including the automatically generated functions). Outside the module, the `BinTree` values are "black boxes" that can be passed around or stored.

If the `BinaryTrees` module is designed and implemented as an information-hiding module as described in Chapter 7, then we also call this an *abstract data type*. We discuss these data abstractions in more detail in Chapter 22.

21.4.3 Defining an alternative binary tree type

Now let's consider a slightly different formulation of a binary tree: a tree in which values are only stored at the leaves.

```
data Tree a = Leaf a | Tree a :^: Tree a
              deriving (Show, Eq)
```

This definition introduces the constructor function name `Leaf` as the constructor for leaves and the infix construction operator "`:^:`" as the constructor for internal nodes of the tree. (A constructor operator symbol must begin with a colon.)

These constructors allow such trees to be defined conveniently. For example, the tree

```
((Leaf 1 :^: Leaf 2) :^: (Leaf 3 :^: Leaf 4))
```

generates a complete binary tree with height 3 and the integers 1, 2, 3, and 4 at the leaves.

Suppose we want a function `fringe`, similar to function `flatten` above, that displays the leaves in a left-to-right order. We can write this as:

```
fringe :: Tree a -> [a]
fringe (Leaf v)  = [v]
fringe (l :^: r) = fringe l ++ fringe r
```

As with `flatten` and `flatten'` above, function `fringe` can also be rendered more efficiently using an accumulating parameter as in the following:

```
fringe' :: Tree a -> [a]
fringe' t = leaves t []
```

```
                where leaves (Leaf v)   = ((:) v)
                      leaves (l :^: r) = leaves l . leaves r
```

Auxiliary function `leaves` builds up the list of leaves from the right using cons.

21.5 Error-handling with `Maybe` and `Either`

Before we examine `Maybe` and `Either`, let's consider a use case.

21.5.1 Handling null references

An *association list* is a list of pairs in which the first component is some *key* (e.g., a string) and the second component is the *value* associated with that key. It is a simple form of a *map* or *dictionary* data structure.

Suppose we have an association list that maps the name of a student (a key) to the name of the student's academic advisor (a value). The following function `lookup'` carries out the search recursively.

```
lookup' :: String -> [(String,String)] -> String
lookup' key ((x,y):xys)
    | key == x   =  y
    | otherwise =  lookup' key xys
```

But what do we do when the key is not in the list (e.g., the list is empty)? How do we define a leg for `lookup' key []` ?

1. Leave the function undefined for that pattern?

   In this case, evaluation will halt with a "non-exhaustive pattern" error message.

2. Put in an explicit `error` call with a custom error message?

3. Return some default value of the advisor such as `"NONE"`?

4. Return a *null reference*?

The first two approaches either halt the entire program or require use of the exception-handling mechanism. However, in any language, both abnormal termination and exceptions should be avoided except in cases in which the program is unable to continue. The lack of an assignment of a student to an advisor is likely not such an extraordinary situation.

Exceptions break referential transparency and, hence, negate many of the advantages of purely functional languages such as Haskell. In addition, Haskell programs can only catch exceptions in `IO` programs (i.e., the outer layers that handle input/output).

The third approach only works when there is some value that is not valid. This is not a very general approach.

The fourth approach, which is not available in Haskell, can be an especially unsafe programming practice. British computing scientist Tony Hoare, who introduced the null reference into the Algol type system in the mid-1960s, calls that his "billion dollar mistake" [17] because it "has led to innumerable errors, vulnerabilities, and system crashes".

What is a safer, more general approach than these?

21.5.2 Introducing Maybe and Either

Haskell includes the union type `Maybe` (from the Prelude and `Data.Maybe`) which can be used to handle such cases.

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord)
```

The `Maybe` algebraic data type encapsulates an optional value. A value of type `Maybe a` either contains a value of type `a` (represented by `Just a`) or it is empty (represented by `Nothing`).

The `Maybe` type is a good way to handle errors or exceptional cases without resorting to an `error` call.

Now we can define a general version of `lookup'` using a `Maybe` return type. (This is essentially function `lookup` from the Prelude.)

```
lookup'' :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup'' key [] =  Nothing
lookup'' key ((x,y):xys)
    | key == x    =  Just y
    | otherwise   =  lookup'' key xys
```

Suppose `advisorList` is an association list pairing students with their advisors and `defaultAdvisor` is the advisor the student should consult if no advisor is officially assigned. We can look up the advisor with a call to `lookup` and then pattern match on the `Maybe` value returned. (Here we use a `case` expression.)

```
whoIsAdvisor :: String -> String
whoIsAdvisor std =
    case lookup std advisorList of
        Nothing   -> defaultAdvisor
        Just prof -> prof
```

The `whoIsAdvisor` function just returns a default value in place of `Nothing`. The function

```
fromMaybe :: a -> Maybe a -> a
```

supported by the `Data.Maybe` library has the same effect. Thus we can rewrite `whoIsAdvisor` as follows:

```
    whoIsAdvisor' std =
        fromMaybe defaultAdvisor $ lookup std advisorList
```

Alternatively, we could use `Data.Maybe` functions such as:

```
    isJust    :: Maybe a -> Bool
    isNothing :: Maybe a -> Bool
    fromJust  :: Maybe a -> a    -- error if Nothing
```

This allows us to rewrite `whoIsAdvisor` as follows:

```
    whoIsAdvisor'' std =
        let ad = lookup std advisorList
        in if isJust ad then fromJust ad else defaultAdvisor
```

If we need more fine-grained error messages, then we can use the union type `Either` defined as follows:

```
    data Either a b = Left a | Right b
                      deriving (Eq, Ord, Read, Show)
```

The `Either a b` type represents values with two possibilities: a `Left a` or `Right b`. By convention, a `Left` constructor usually contains an error message and a `Right` constructor a *correct* value.

As with `fromMaybe`, we can use similar `fromRight` and `fromLeft` functions from the `Data.Either` library to extract the `Right` or `Left` values or to return a default value when the value is represented by the other constructor.

```
    fromLeft  :: a -> Either a b -> a
    fromRight :: b -> Either a b -> b
```

Library module `Data.Either` also includes functions to query for the presence of the two constructors.

```
    isLeft    :: Either a b -> Bool
    isRight   :: Either a b -> Bool
```

21.5.3 Considering other languages

Most recently designed languages include a *maybe* or *option* type [33]. Scala [20,25] has an `Option` case class [5:4,10], Rust [19,24] has an `Option` enum, and Swift has an `Optional` class, all of which are similar to Haskell's `Maybe`. The functional languages Idris [2,3], Elm [13,15], and PureScript [16,21] also have Haskell-like `Maybe` algebraic data types.

The concept of *nullable type* [31] is closely related to the option type. Several older languages support this concept (e.g., `Optional` in Java 8, `None` in Python [22,23], and `?` type annotations in C#).

When programming in an object-oriented language that does not provide an option/maybe type, a programmer can often use the *Null Object design pattern*

[26,32,34] to achieve a similar result. This well-known pattern seeks to "encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior" [26]. That is, the object must be of the correct type. It must be possible to apply all operations on that type to the object, but the operations should have neutral behaviors, with no side effects. The null object should actively do nothing!

21.6 What Next?

This chapter (21) added Haskell's algebraic data types to our programming toolbox. Chapter 22 sharpens the data abstraction tools introduced in Chapter 7 by using algebraic data types from this chapter. Chapter 23 adds type classes and overloading to the toolbox.

The remainder of this chapter includes a number of larger exercises and projects.

21.7 Chapter Source Code

The Haskell source module for this chapter is in file `AlgDataTypes.hs`.

21.8 Exercises

1. For trees of type `Tree`, implement a tree-folding function similar to `treeFold`.

2. For trees of type `BinTree`, implement a version of `treeFold` that uses an accumulating parameter. (Hint: `foldl`.)

3. In a binary search tree all values in the left subtree of a node are less than the value at the node and all values in the right subtree are greater than the value at the node.

   Given binary search trees of type `BinTree`, implement the following Haskell functions:

   a. `makeTree` that takes a list and returns a perfectly balanced (i.e., minimal height) `BinTree` such that `flatten (makeTree xs) = sort xs`. Prelude function `sort` returns its argument rearranged into ascending order.

   b. `insertTree` that takes an element and a `BinTree` and returns the `BinTree` with the element inserted at an appropriate position.

   c. `elemTree` that takes an element and a `BinTree` and returns `True` if the element is in the tree and `False` otherwise.

   d. `heightTree` that takes a `BinTree` and returns its height. Assume that height means the number of levels in the tree. (A tree consisting of exactly one node has a height of `1`.)

e. `mirrorTree` that takes a `BinTree` and returns its mirror image. That is, it takes a tree and returns the tree with the left and right subtrees of every node swapped.

f. `mapTree` that takes a function and a `BinTree` and returns the `BinTree` of the same shape except each node's value is computed by applying the function to the corresponding value in the input tree.

g. `showTree` that takes a `BinTree` and displays the tree in a parenthesized, left-to-right, in-order traversal form. (That is, the traversal of a tree is enclosed in a pair of parentheses, with the traversal of the left subtree followed by the traversal of the right subtree.)

Extend the package to support both insertion and deletion of elements. Keep the tree balanced using a technique such the AVL balancing algorithm.

4. Implement the package of functions described in the previous exercise for the data type `Tree`.

5. Each node of a general (i.e., multiway) tree consists of a label and a list of (zero or more) subtrees (each a general tree). We can define a general tree data type in Haskell as follows:

```
data Gtree a = Node a [Gtree a]
```

For example, tree `(Node 0 [ ])` consists of a single node with label `0`; a more complex tree `Node 0 [Node 1 [ ], Node 2 [ ], Node 3 []]` consists of root node with three single-node subtrees.

Implement a "map" function for general trees, i.e., write Haskell function

```
mapGtree :: (a -> b) -> Gtree a -> Gtree b
```

that takes a function and a `Gtree` and returns the `Gtree` of the same shape such that each label is generated by applying the function to the corresponding label in the input tree.

6. We can introduce a new Haskell type for the natural numbers (i.e., nonnegative integers) with the statement

```
data Nat = Zero | Succ Nat
```

where the constructor `Zero` represents the value 0 and constructor `Succ` represents the "successor function" from mathematics. Thus `(Succ Zero)` denotes 1, `(Succ (Succ Zero))` denotes 2, and so forth. Implement the following Haskell functions.

a. `intToNat` that takes a nonnegative `Int` and returns the equivalent `Nat`, for example, `intToNat 2` returns `Succ (Succ Zero)`.

b. `natToInt` that takes a `Nat` and returns the equivalent value of type `Int`, for example, `natToInt Succ (Succ Zero)` returns `2`.

c. `addNat` that takes two `Nat` values and returns their sum as a `Nat`. This function cannot use integer addition.

d. `mulNat` that takes two `Nat` values and returns their product as a `Nat`. This function cannot use integer multiplication or addition.

e. `compNat` that takes two `Nat` values and returns the value -1 if the first is less than the second, 0 if they are equal, and 1 if the first is greater than the second. This function cannot use the integer comparison operators.

7. Consider the following Haskell data type for representing sequences (i.e., lists):

```
data Seq a = Nil | Att (Seq a) a
```

`Nil` represents the empty sequence. `Att xz y` represents the sequence in which *last element* `y` is "attached" at the right end of the *initial sequence* `xz`.

Note that `Att` is similar to the ordinary "cons" (`:`) for Haskell lists except that elements are attached at the opposite end of the sequences. `(Att (Att (Att Nil 1) 2) 3)` represents the same sequence as the ordinary list `(1:(2:(3:[])))`.

Implement Haskell functions for the following operations on type `Seq`. The operations are analogous to the similarly named operations on the built-in Haskell lists.

a. `lastSeq` takes a nonempty `Seq` and returns its last (i.e., rightmost) element.

b. `initialSeq` takes a nonempty `Seq` and returns its initial sequence (i.e., sequence remaining after the last element removed).

c. `lenSeq` takes a `Seq` and returns the number of elements that it contains.

d. `headSeq` takes a nonempty `Seq` and returns its head (i.e., leftmost) element.

e. `tailSeq` takes a nonempty `Seq` and returns the `Seq` remaining after the head element is removed.

f. `conSeq` that takes an element and a `Seq` and returns a `Seq` with the argument element as its head and the `Seq` argument as its tail.

g. `appSeq` takes two arguments of type `Seq` and returns a `Seq` with the second argument appended after the first.

h. `revSeq` takes a `Seq` and returns the `Seq` with the same elements in reverse order.

i. `mapSeq` takes a function and a `Seq` and returns the `Seq` resulting from applying the function to each element of the sequence in turn.

j. `filterSeq` that takes a predicate and a `Seq` and returns the `Seq` containing only those elements that satisfy the predicate.

k. `listToSeq` takes an ordinary Haskell list and returns the `Seq` with the same values in the same order (e.g., `headSeq (listToSeq xs) = head xs` for nonempty `xs`.)

l. `seqToList` takes a `Seq` and returns the ordinary Haskell list with the same values in the same order (e.g., `head (seqToList xz) = headSeq xz` for nonempty `xz`.)

8. Consider the following Haskell data type for representing sequences (i.e., lists):

```
data Seq a = Nil | Unit a | Cat (Seq a) (Seq a)
```

The constructor `Nil` represents the empty sequence; `Unit` represents a single-element sequence; and `Cat` represents the "concatenation" (i.e., append) of its two arguments, the second argument appended after the first.

Implement Haskell functions for the following operations on type `Seq`. The operations are analogous to the similarly named operations on the built-in Haskell lists. (Do not convert back and forth to lists.)

a. `toSeq` that takes a list and returns a corresponding `Seq` that is balanced.

b. `fromSeq` that takes a `Seq` and returns the corresponding list.

c. `appSeq` that takes two arguments of type `Seq` and returns a `Seq` with the second argument appended after the first.

d. `conSeq` that takes an element and a `Seq` and returns a `Seq` with the argument element as its head and the `Seq` argument as its tail.

e. `lenSeq` that takes a `Seq` and returns the number of elements that it contains.

f. `revSeq` that takes a `Seq` and returns a `Seq` with the same elements in reverse order.

g. `headSeq` that takes a nonempty `Seq` and returns its head (i.e., leftmost or front) element. (Be careful!)

h. `tailSeq` that takes a nonempty `Seq` and returns the `Seq` remaining after the head is removed.

i. `normSeq` that takes a `Seq` and returns a `Seq` with unnecessary embedded `Nil` values removed. (For example, `normSeq (Cat (Cat Nil (Unit 1)) Nil)` returns `(Unit 1)`.)

j. `eqSeq` that takes two `Seq` "trees" and returns `True` if the sequences of values are equal and returns `False` otherwise. Note that two `Seq` "trees" may be structurally different yet represent the same sequence of values.

   For example, `(Cat Nil (Unit 1))` and `(Cat (Unit 1) Nil)` have the same sequence of values (i.e., `[1]`). But `(Cat (Unit 1) (Unit 2))` and `(Cat (Unit 2) (Unit 1))` do not represent the same sequence of values (i.e., `[1,2]` and `[2,1]`, respectively).

   Also `(Cat (Cat (Unit 1) (Unit 2)) (Unit 3))` has the same sequence of values as `(Cat (Cat (Unit 1) (Unit 2)) (Unit 3))` (i.e., `[1,2,3]`).

In general what are the advantages and disadvantages of representing lists this way?

21.9 Carrie's Candy Bowl Project

21.9.1 Problem description and initial design

Carrie, the Department's Administrative Assistant, has a candy bowl on her desk. Often she fills this bowl with candy, but the contents are quickly consumed by students, professors, and staff members. At a particular point in time, the candy bowl might contain several different kinds of candy with one or more pieces of each kind or it might be empty. Over time, the kinds of candy in the bowl varies.

In this project, we model the candy, the candy bowl, and the "operations" that can be performed on the bowl and develop it as a Haskell module.

What about the candy?

In general, we want to be able to identify how many pieces of candy we have of a particular kind (e.g., we may have two Snickers bars and fourteen Hershey's Kisses) but do not need to distinguish otherwise between the pieces. So distinct identifiers for the different kinds of candy should be sufficient.

We can represent the different kinds of candy in several different ways. We could use strings, integer codes, the different values of an enumerated type, etc. In different circumstances, we might want to use different representations.

Thus we model the kinds of candy to be a polymorphic parameter of the candy bowl. However, we can contrain the polymorphism on the kinds of candy to be a Haskell type that can be compared for equality (i.e., in class `Eq`) and converted to a string so that it can be displayed (i.e., in class `Show`).

What about the candy bowl itself?

A candy bowl is some type of collection of pieces of candy with several possible representations. We could use a list (either unordered) of the pieces of candy, an association list (unordered or ordered) pairing the kinds of candy with the numbers of pieces of each, a `Data.Map` structure (from the Haskell library), or some other data structure.

Thus we want to allow the developers of the candy bowl to freely choose whatever representation they wish or perhaps to provide several different implementations with the same interface. We will leave this hidden inside the Haskell module that implements an abstract data type.

Thus, a Haskell module that implements the candy bowl can define a polymophic algebraic data type `CandyBowl a` and export its name, but not export the implementation details (i.e., the constructors) of the type. For example, a represenation built around a list of kinds of candy could be defined as:

```
data CandyBowl a = Bowl [a]
```

Or a representation using an association list can be defined as:

```
data CandyBowl a = Bowl [(a,Int)]
```

Thus, to export the `CandyBowl` but hide the details of the representation, the module would have a header such as:

```
module CarrieCandyBowl
    ( CandyBowl(..), -- function names exported
    )
where  -- implementation details of type and functions
```

Some of the possible representations require the ability to order the types of candy in some way. Thus, we further constrain the polymorphic type parameter to class `Ord` instead of simply `Eq`. (Above, we also constrained it to class `Show`.)

21.9.2 Carrie's Candy Bowl project exercises

Your task for this project to develop a Haskell module `CarrieCandyBowl` (in a file `CarrieCandyBowl.hs`), as described above. You must choose an appropriate internal representation for the data type `CandyBowl` and implement the public operations (functions) defined below. In addition to exporting the public functions and data type name, the module may contain whatever other internal data and function definitions needed for theimplemenation.

An initial Haskell source code for this project is in file `CarrieCandyBowl_skeleton.hs`.

You may use a function you have completed to implement other functions in the list (as long as you do not introduce circular definitions).

1. `newBowl :: (Ord a,Show a) => CandyBowl a`
   creates a new empty candy bowl.

2. `isEmpty :: (Ord a,Show a) => CandyBowl a -> Bool`
   returns `True` if and only if the bowl is empty.

3. `putIn :: (Ord a,Show a) => CandyBowl a -> a -> CandyBowl a`
   adds one piece of candy of the given kind to the bowl.

   For example, if we use strings to represent the kinds, then

   ```
   putIn bowl "Kiss"
   ```

   adds one piece of candy of kind `"Kiss"` to the `bowl`.

4. `has :: (Ord a,Show a) => CandyBowl a -> a -> Bool`
   returns `True` if and only if one or more pieces of the given kind of candy is in the bowl.

5. `size :: (Ord a,Show a) => CandyBowl a -> Int`
   returns the total number of pieces of candy in the bowl (regardless of kind).

6. `howMany :: (Ord a,Show a) => CandyBowl a -> a -> Int`
   returns the count of the given kind of candy in the bowl.

7. `takeOut :: (Ord a,Show a) => CandyBowl a -> a -> Maybe (CandyBowl a)`
   attempts to remove one piece of candy of the given kind from the bowl (so it can be eaten). If the bowl contains a piece of the given kind, the function returns the value `Just bowl`, where `bowl` is the bowl with the piece removed. If the bowl does not contain such a piece, it returns the value `Nothing`.

8. `eqBowl :: (Ord a,Show a) => CandyBowl a -> CandyBowl a -> Bool`
   returns `True` if and only if the two bowls have the same contents (i.e., the same kinds of candy and the same number of pieces of each kind).

9. `inventory :: (Ord a,Show a) => CandyBowl a -> [(a,Int)]`
   returns a Haskell list of pairs `(k,n)`, where each kind `k` of candy in the bowl occurs once in the list with `n > 0`. The list should be arranged in *ascending order* by kind.

   For example, if there are two `"Snickers"` and one `"Kiss"` in the bowl, the list returned would be `[("Kiss",1),("Snickers",2)]`.

10. `restock :: (Ord a,Show a) => [(a,Int)] -> CandyBowl a`
    creates a new bowl such that for any `bowl`:

    ```
    eqBowl (restock (inventory bowl)) bowl  ==  True
    ```

11. `combine :: (Ord a,Show a) => CandyBowl a -> CandyBowl a -> CandyBowl a`
    pours the two bowls together to form a new “larger” bowl.

12. `difference :: (Ord a,Show a) => CandyBowl a -> CandyBowl a -> CandyBowl a`
    returns a bowl containing the pieces of candy in the first bowl that are not in the second bowl.

    For example, if the first bowl has four `"Snickers"` and the second has one `"Snickers"`, then the result will have three `"Snickers"`.

13. `rename :: (Ord a,Show a) => CandyBowl a -> (a -> b) -> CandyBowl b` takes a bowl and a renaming function, applies the renaming function to all the kind values in the bowl, and returns the modified bowl.

    For example, for some mysterious reason, we might want to reverse the strings for the kind names: `f xs = reverse xs`. Thus `"Kiss"` would become `"ssiK"`. Then `rename f bowl` would do the reversing of all the names.

21.9.3 Candy Bowl alternative exercises

TODO: Maybe specify reimplentations with a different data rep, perhaps requiring a map.

21.10 Sandwich DSL Project

21.10.1 Project Introduction

Few computer science graduates will design and implement a general-purpose programming language during their careers. However, many graduates will design and implement—and all likely will use—special-purpose languages in their work.

These special-purpose languages are often called *domain-specific languages* (or DSLs) [11]. (For more discussion of the DSL concepts, terminology, and techniques, see the introductory chapter of the Notes on Domain-Specific Languages [11].)

In this project, we design and implement a simple *internal DSL* [11]. This DSL describes simple "programs" using a set of Haskell algebraic data types. We express a program as an *abstract syntax tree* (AST) [11] using the DSLs data types.

In this project, we first build a package of functions for creating and manipulating the abstract syntax trees. We then extend the package to translate the abstract syntax trees to a sequence of instructions for a simple "machine".

21.10.2 Developing the Sandwich DSL

Suppose Emerald de Gassy, the owner of the Oxford-based catering business Deli-Gate, hires us to design a domain-specific language (DSL) for describing sandwich platters. The DSL scripts will direct Deli-Gate's robotic kitchen appliance SueChef (Sandwich and Utility Electronic Chef) to assemble platters of sandwiches.

In discussing the problem with Emerald and the Deli-Gate staff, we discover the following:

- A sandwich platter consists of zero or more sandwiches. (Zero? Why not! Although a platter with no sandwiches may not be a useful, or profitable, case, there does not seem to be any harm in allowing this degenerate case. It may simplify some of the coding and representation.)

- Each sandwich consists of layers of ingredients.
- The categories of ingredients are breads, meats, cheeses, vegetables, and condiments.
- Available breads are white, wheat, and rye.
- Available meats are turkey, chicken, ham, roast beef, and tofu. (Okay, tofu is not a meat, but it is a good protein source for those who do not wish to eat meat. This is a college town after all.)
- Available cheeses are American, Swiss, jack, and cheddar.
- Available vegetables are tomato, lettuce, onion, and bell pepper.
- Available condiments are mayo, mustard, relish, and Tabasco. (Of course, this being the South, the mayo is Blue Plate Mayonnaise and the mustard is a Creole mustard.)

Let's define this as an internal DSL—in particular, by using a relatively *deep embedding* [11].

What is a sandwich? ... Basically, it is a stack of ingredients.

Should we require the sandwich to have a bread on the bottom? ... Probably. ... On the top? Maybe not, to allow "open-faced" sandwiches. ... What can the SueChef build? ... We don't know at this point, but let's assume it can stack up any ingredients without restriction.

For simplicity and flexibility, let's define a Haskell data type `Sandwich` to model sandwiches. It wraps a possibly empty list of ingredient layers. *We assume the head of the list to be the layer at the top of the sandwich.* We derive `Show` so we can display sandwiches.

```
data Sandwich = Sandwich [Layer]
                deriving Show
```

Note: In this project, we use the same name for an algebraic data type and its only constructor. Above the `Sandwich` after `data` defines a type and the one after the "`=`" defines the single constructor for that type.

Data type `Sandwich` gives the specification for a sandwich. When "executed" by the SueChef, it results in the assembly of a sandwich that satisfies the specification.

As defined, the `Sandwich` data type does not require there to be a bread in the stack of ingredients. However, we add function `newSandwich` that starts a sandwich with a bread at the bottom and a function `addLayer` that adds a new ingredient to the top of the sandwich. We leave the implementation of these functions as exercises.

```
newSandwich :: Bread -> Sandwich
addLayer    :: Sandwich -> Layer -> Sandwich
```

Ingredients are in one of five categories: breads, meats, cheeses, vegetables, and condiments.

Because both the categories and the specific type of ingredient are important, we choose to represent both in the type structures and define the following types. A value of type `Layer` represents a single ingredient. Note that we use names such as `Bread` both as a constructor of the `Layer` type and the type of the ingredients within that category.

```
data Layer     = Bread Bread          | Meat Meat
               | Cheese Cheese        | Vegetable Vegetable
               | Condiment Condiment
                 deriving (Eq,Show)

data Bread     = White | Wheat | Rye
                 deriving (Eq, Show)

data Meat      = Turkey | Chicken | Ham | RoastBeef | Tofu
                 deriving (Eq, Show)

data Cheese    = American | Swiss | Jack | Cheddar
                 deriving (Eq, Show)

data Vegetable = Tomato | Onion | Lettuce | BellPepper
                 deriving (Eq, Show)

data Condiment = Mayo | Mustard | Ketchup | Relish | Tabasco
                 deriving (Eq, Show)
```

We need to be able to compare ingredients for equality and convert them to strings. Because the automatically generated default definitions are appropriate, we derive both classes `Show` and `Eq` for these ingredient types.

We do not derive `Eq` for `Sandwich` because the default element-by-element equality of lists does not seem to be the appropriate equality comparison for sandwiches.

To complete the model, we define type `Platter` to wrap a list of sandwiches.

```
data  Platter = Platter [Sandwich]
                deriving Show
```

We also define functions `newPlatter` to create a new `Platter` and `addSandwich` to add a sandwich to the `Platter`. We leave the implementation of these functions as exercises.

```haskell
newPlatter  :: Platter
addSandwich :: Platter -> Sandwich -> Platter
```

21.10.3 Sandwich DSL exercise set A

Please put these functions in a Haskell module `SandwichDSL` (in a file named `SandwichDSL`.) You may use functions defined earlier in the exercises to implement those later in the exercises.

1. Define and implement the Haskell functions `newSandwich`, `addLayer`, `newPlatter`, and `addSandwich` described above.

2. Define and implement the Haskell query functions below that take an ingredient (i.e., `Layer`) and return `True` if and only if the ingredient is in the specified category.

   ```haskell
   isBread     :: Layer -> Bool
   isMeat      :: Layer -> Bool
   isCheese    :: Layer -> Bool
   isVegetable :: Layer -> Bool
   isCondiment :: Layer -> Bool
   ```

3. Define and implement a Haskell function `noMeat` that takes a sandwich and returns `True` if and only if the sandwich contains no meats.

   ```haskell
   noMeat :: Sandwich -> Bool
   ```

4. According to a proposed City of Oxford ordinance, in the future it may be necessary to assemble all sandwiches in *Oxford Standard Order (OSO)*: a slice of bread on the bottom, then zero or more meats layered above that, then zero or more cheeses, then zero or more vegetables, then zero or more condiments, and then a slice of bread on top. The top and bottom slices of bread must be of the same type.

   Define and implement a Haskell function `inOSO` that takes a sandwich and determines whether it is in OSO and another function `intoOSO` that takes a sandwich and a default bread and returns the sandwich with the same ingredients ordered in OSO.

   ```haskell
   inOSO   :: Sandwich -> Bool
   intoOSO :: Sandwich -> Bread -> Sandwich
   ```

   Hint: Remember Prelude functions like `dropWhile`.

   Note: It is impossible to rearrange the layers into OSO if the sandwich does not include exactly two breads of the same type. If the sandwich does not include any breads, then the default bread type (second argument) should be specified for both. If there is at least one bread, then the bread type nearest the *bottom* can be chosen for both top and bottom.

5. Suppose we store the current prices of the sandwich ingredients in an association list with the following type synonym:

   ```haskell
   type PriceList = [(Layer,Int)]
   ```

   Assuming that the price for a sandwich is base price plus the sum of the prices of the individual ingredients, define and implement a Haskell function `priceSandwich` that takes a price list, a base price, and a sandwich and returns the price of the sandwich.

   ```haskell
   priceSandwich :: PriceList -> Int -> Sandwich -> Int
   ```

   Hint: Consider using the `lookup` function from the Prelude. The library `Data.Maybe` may also include helpful functions.

   Use the following price list as a part of your testing:

   ```haskell
   prices = [ (Bread White, 20), (Bread Wheat, 30),
              (Bread Rye, 30),
              (Meat Turkey, 100), (Meat Chicken, 80),
              (Meat Ham, 120), (Meat RoastBeef, 140),
              (Meat Tofu, 50),
              (Cheese American, 50), (Cheese Swiss, 60),
              (Cheese Jack, 60), (Cheese Cheddar, 60),
              (Vegetable Tomato, 25), (Vegetable Onion, 20),
              (Vegetable Lettuce, 20), (Vegetable BellPepper,25),
              (Condiment Mayo, 5), (Condiment Mustard, 4),
              (Condiment Ketchup, 4), (Condiment Relish, 10),
              (Condiment Tabasco, 5)
            ]
   ```

6. Define and implement a Haskell function `eqSandwich` that compares two sandwiches for equality.

   What does equality mean for sandwiches? Although the definition of equality could differ, you can use "bag equality". That is, two sandwiches are equal if they have the same number of layers (zero or more) of each ingredient, regardless of the order of the layers.

   ```haskell
   eqSandwich :: Sandwich -> Sandwich -> Bool
   ```

   Hint: The "sets" operations in library `Data.List` might be helpful

7. Give the Haskell declaration needed to make `Sandwich` an instance of class `Eq`. You may use `eqSandwich` if applicable.

21.10.4 Compiling the program for the SueChef controller

In this section, we look at compiling the `Platter` and `Sandwich` descriptions to issue a sequence of commands for the SueChef's controller.

The SueChef supports the special instructions that can be issued in sequence to its controller. The data type `SandwichOp` below represents the instructions.

```haskell
data SandwichOp = StartSandwich    | FinishSandwich
                | AddBread Bread   | AddMeat Meat
                | AddCheese Cheese | AddVegetable Vegetable
                | AddCondiment Condiment
                | StartPlatter | MoveToPlatter | FinishPlatter
                  deriving (Eq, Show)
```

We also define the type `Program` to represent the sequence of commands resulting from compilation of a `Sandwich` or `Platter` specification.

```haskell
data Program = Program [SandwichOp]
               deriving Show
```

The flow of a program is given by the following pseudocode:

```
StartPlatter
for each sandwich needed
    StartSandwich
    for each ingredient needed
        Add ingredient on top
    FinishSandwich
    MoveToPlatter
FinishPlatter
```

Consider a sandwich defined as follows:

```haskell
Sandwich [ Bread Rye, Condiment Mayo, Cheese Swiss,
           Meat Ham, Bread Rye ]
```

The corresponding sequence of SueChef commands would be the following:

```haskell
[ StartSandwich, AddBread Rye, AddMeat Ham, AddCheese Swiss,
  AddCondiment Mayo, AddBread Rye, FinishSandwich, MoveToPlatter ]
```

21.10.5 Sandwich DSL exercise set B

Add the following functions to the module `SandwichDSL` developed in the Sandwich DSL Project exercise set A.

1. Define and implement a Haskell function `compileSandwich` to convert a sandwich specification into the sequence of SueChef commands to assemble the sandwich.

    ```haskell
    compileSandwich :: Sandwich -> [SandwichOp]
    ```

2. Define and implement a Haskell function `compile` to convert a platter specification into the sequence of SueChef commands to assemble the sandwiches on the platter.

```
        compile :: Platter -> Program
```

21.10.6 Sandwich DSL source code

The Haskell source code for this project is in file:

- `SandwichDSL_base.hs`

21.11 Exam DSL Project

21.11.1 Project introduction

Few computer science graduates will design and implement a general-purpose programming language during their careers. However, many graduates will design and implement—and all likely will use—special-purpose languages in their work.

These special-purpose languages are often called *domain-specific languages* (or DSLs) [11]. (For more discussion of the DSL concepts, terminology, and techniques, see the introductory chapter of the Notes on Domain-Specific Languages [11].)

In this project, we design and implement a simple *internal DSL* [11]. This DSL describes simple "programs" using a set of Haskell algebraic data types. We express a program as an *abstract syntax tree* (AST) [11] using the DSL's data types.

The package first builds a set of functions for creating and manipulating the abstract syntax trees for the exams. It then extends the package to translate the abstract syntax trees to HTML.

21.11.2 Developing the Exam DSL

Suppose Professor Harold Pedantic decides to create a DSL to encode his (allegedly vicious) multiple choice examinations. Since his course uses Haskell to teach programming language organization, he wishes to implement the language processor in Haskell. Professor Pedantic is too busy to do the task himself. He is also cheap, so he assigns us, the students in his class, the task of developing a prototype.

In the initial prototype, we do not concern ourselves with the concrete syntax of the Exam DSl. We focus on design of the AST as a Haskell algebraic data type. We seek to design a few useful functions to manipulate the AST and output an exam as HTML.

First, let's focus on multiple-choice questions. For this prototype, we can assume a question has the following components:

- the text of the question

- a group of several choices for the answer to the question, exactly one of which should be be a correct answer to the question
- a group of tags identifying topics covered by the question

Let's define this as an internal DSL—in particular, by using a relatively *deep embedding* [11].

We can state a question using the Haskell data type `Question`, which has a single constructor `Ask`. It has three components—a list of applicable topic tags, the text of the question, and a list of possible answers to the question.

```
type QText    = String
type Tag      = String
data Question = Ask [Tag] QText [Choice] deriving Show
```

We use the type `QText` to describe the text of a question. We also use the type `Tag` to describe the topic tags we can associate with a question.

We can then state a possible answer to the question using the data type `Choice`, which has a single constructor `Answer`. It has two components—the text of the answer and a Boolean value that indicates whether this is a correct answer to the question (i.e., `True`) or not.

```
type AText    = String
data Choice   = Answer AText Bool deriving (Eq, Show)
```

As above, we use the type `AText` to describe the text of an answer.

For example, we could encode the question "Which of the following is a required course?" as follows.

```
Ask ["curriculum"]
    "Which of the following is a required course?"
    [ Answer "CSci 323" False,
      Answer "CSci 450" True,
      Answer "CSci 525" False ]
```

The example has a single topic tag `"curriculum"` and three possible answers, the second of which is correct.

We can develop various useful functions on these data types. Most of these are left as exercises.

For example, we can define a function `correctChoice` that takes a `Choice` and determines whether it is marked as a correct answer or not.

```
correctChoice :: Choice -> Bool
```

We can also define function `lenQuestion` that takes a question and returns the number of possible answers are given. This function has the following signature.

```
lenQuestion :: Question -> Int
```

We can then define a function to check whether a question is valid. That is, the question must have:

- a non-nil text
- at least 2 and no more than 10 possible answers
- exactly one correct answer

It has the type signature.

```
validQuestion :: Question -> Bool
```

We can also define a function to determine whether or not a question has a particular topic tag.

```
hasTag :: Question -> Tag -> Bool
```

To work with our lists of answers (and other lists in our program), let's define function `eqBag` with the following signature.

```
eqBag :: Eq a => [a] -> [a] -> Bool
```

This is a "bag equality" function for two polymorphic lists. That is, the lists are collections of elements that can be compared for equality and inequality, but not necessarily using ordered comparisons. There may be elements repeated in the list.

Now, what does it mean for two questions to be equal?

For our prototype, we require that the two questions have the same question text, the same collection of tags, and the same collection of possible answers with the same answer marked correct. However, we do not require that the tags or possible answers appear in the same order.

We note that type `Choice` has a derived instance of class `Eq`. Thus we can give an `instance` definition to make `Question` an instance of class `Eq`.

```
instance Eq Question where
-- fill in the details
```

Now, let's consider the examination as a whole. It consists of a title and a list of questions. We thus define the data type `Exam` as follows.

```
type Title = String
data Exam = Quiz Title [Question] deriving Show
```

We can encode an exam with two questions as follows.

```
Quiz "Curriculum Test" [
     Ask ["curriculum"]
         "Which one of the following is a required course?"
         [ Answer "CSci 323" False,
           Answer "CSci 450" True,
           Answer "CSci 525" False ],
     Ask ["language","course"]
         "What one of the following languages is used in CSci 450?"
         [ Answer "Lua" False,
           Answer "Elm" False,
           Answer "Haskell" True ]
     ]
```

We can define function `selectByTags` selects questions from an exam based on the occurrence of the specified topic tags.

```
selectByTags :: [Tag] -> Exam -> Exam
```

The function application `selectByTags tags exam` takes a list of zero or more `tags` and an `exam` and returns an exam with only those questions in which at least one of the given `tags` occur in a `Question`'s tag list.

We can define function `validExam` that takes an exam and determines whether or not it is valid. It is valid if and only if all questions are valid. The function has the following signature.

```
validExam :: Exam -> Bool
```

To assist in grading an exam, we can also define a function `makeKey` that takes an exam and creates a list of `(number,letter)` pairs for all its questions. In a pair, `number` is the problem number, a value that starts with `1` and increases for each problem in order. Similarly, `letter` is the answer identifier, an uppercase alphabetic character that starts with `A` and increases for each choice in order. The function returns the tuples arranged by increasing problem number.

The function has the following signature.

```
makeKey :: Exam -> [(Int,Char)]
```

For the example exam above, `makeKey` should return `[(1,'B'),(2,'C')]`.

21.11.3 Exam DSL exercise set A

Define the following functions in a module named `ExamDSL` (in a file named `ExamDSL.hs`).

1. Develop function `correctChoice :: Choice -> Bool` as defined above.
2. Develop function `lenQuestion :: Question -> Int` as defined above.
3. Develop function `validQuestion :: Question -> Bool` as defined above.
4. Develop function `hasTag :: Question -> Tag -> Bool` as defined above.
5. Develop function `eqBag :: Eq a => [a] -> [a] -> Bool` as defined above.

6. Give an `instance` declaration to make data type `Question` an instance of class `Eq`.
7. Develop function `selectByTags :: [Tag] -> Exam -> Exam` as defined above.
8. Develop function `validExam :: Exam -> Bool` as defined above.
9. Develop function `makeKey :: Exam -> [(Int,Char)]` as defined above.

21.11.4 Outputting the Exam as HTML

Professor Pedantic wants to take an examination expressed with the Exam DSL, as described above, and output it as HTML.

Again, consider the following `Exam` value.

```
Quiz "Curriculum Test" [
     Ask ["curriculum"]
         "Which one of the following courses is required?"
         [ Answer "CSci 323" False,
           Answer "CSci 450" True,
           Answer "CSci 525" False ],
     Ask ["language","course"]
         "What one of the following is used in CSci 450?"
         [ Answer "Lua" False,
           Answer "Elm" False,
           Answer "Haskell" True ]
     ]
```

We want to convert the above to the following HTML.

```
<html lang="en">
<body>
<h1>Curriculum Test</h1>
<ol type="1">
<li>Which one of the following courses is required?
<ol type="A">
<li>CSci 323</li>
<li>CSci 450</li>
<li>CSci 525</li>
</ol>
</li>
<li>What one of the following is used in CSci 450?
<ol type="A">
<li>Lua</li>
<li>Elm</li>
<li>Haskell</li>
</ol>
```

```
</li>
</ol>
</body>
</html>
```

This would render in a browser something like the following.

Curriculum Test

1. Which one of the following courses is required?
   A. CSci 323
   B. CSci 450
   C. CSci 525
2. What one of the following is used in CSci 450?
   A. Lua
   B. Elm
   C. Haskell

Professor Pedantic developed a module of HTML template functions named `SimpleHTML` to assist us in this process. (See file `SimpleHTML.hs`.)

A function application `to_html lang content` wraps the `content` (HTML in a string) inside a pair of HTML tags `<html>` and `</html>` with `lang` attribute set to `langtype`, defaulting to `English` (i.e., `"en"`). This function and the data types are defined in the following.

```
type HTML     = String
data LangType = English | Spanish | Portuguese | French
                deriving (Eq, Show)
langmap = [ (English,"en"), (Spanish,"es"), (Portuguese,"pt"),
            (French,"fr") ]

to_html :: LangType -> HTML -> HTML
to_html langtype content =
    "<html lang=\"" ++ lang ++ "\">" ++ content ++ "</html>"
        where lang = case lookup langtype langmap of
                         Just l  -> l
                         Nothing -> "en"
```

For the above example, the `to_html` function generates the the outer layer:

```
<html lang="en"> ... </html>
```

Function application `to_body content` wraps the `content` inside a pair of HTML tags `<body>` and `</body>`.

```
to_body :: HTML -> HTML
to_body content = "<body>" ++ content ++ "</body>"
```

Function application `to_heading level title` wraps string `title` inside a pair of HTML tags `<hN>` and `</hN>` where `N` is in the range 1 to 6. If `level` is outside this range, it defaults to the nearest valid value.

```
to_heading:: Int -> String -> HTML
to_heading level title = open ++ title ++ close
    where lev   = show (min (max level 1) 6)
          open  = "<h"  ++ lev ++ ">"
          close = "</h" ++ lev ++ ">"
```

Function application `to_list listtype content` wraps the `content` inside a pair of HTML tags `<ul>` and `</ul>` or `<ol>` and `</ol>`. For `<ol>` tags, it sets the `type` attribute based on the value of the `listtype` argument.

```
data ListType = Decimal | UpRoman | LowRoman
              | UpLettered | LowLettered | Bulleted
                deriving (Eq, Show)

to_list :: ListType -> HTML -> HTML
to_list listtype content = open ++ content ++ close
    where
        (open,close) =
            case listtype of
                Decimal     -> ("<ol type=\"1\">", "</ol>")
                UpRoman     -> ("<ol type=\"I\">", "</ol>")
                LowRoman    -> ("<ol type=\"i\">", "</ol>")
                UpLettered  -> ("<ol type=\"A\">", "</ol>")
                LowLettered -> ("<ol type=\"a\">", "</ol>")
                Bulleted    -> ("<ul>",            "</ul>")
```

Finally, function application `to_li content` wraps the `content` inside a pair of HTML tags `<li>` and `</li>`.

```
to_li :: HTML -> HTML
to_li content = "<li>" ++ content ++ "</li>"
```

By importing the `SimpleHTML` module, we can now develop functions to output an `Exam` as HTML.

If we start at the leaves of the `Exam` AST (i.e., from the `Choice` data type), we can define a function `choice2html` function as follows in terms of `to_li`.

```
choice2html :: Choice -> HTML
choice2html (Answer text _) = to_li text
```

Using `choice2html` and the `SimpleHTML` module, we can define `question2html` with the following signature.

```
question2html :: Question -> HTML
```

Then using `question2html` and the `SimpleHTML` module, we can define `question2html` with the following signature.

```
exam2html :: Exam -> HTML
```

Note: These two functions should add newline characters to the HTML output so that they look like the examples at the beginning of the "Outputting the Exam" section. Similarly, it should not output extra spaces. This both makes the string output more readable and makes it possible to grade the assignment using automated testing.

For example, the output of `question2html` for the first `Question` in the example above should appear as the following when printed with the `putStr` input-output command.

```
<li>Which one of the following courses is required?
<ol type="A">
<li>CSci 323</li>
<li>CSci 450</li>
<li>CSci 525</li>
</ol>
```

In addition, you may want to output the result of `exam2html` to a file to see how it displays in a browser a particular `exam`.

```
writeFile "output.html" $ exam2html exam
```

21.11.5 Exam DSL project exercise set B

Add the following functions to the module `ExamDSL` developed in the Exam DSL Project exercise set A.

1. Develop function `question2html :: Question -> HTML` as defined above.
2. Develop function `exam2html :: Exam -> HTML` as defined above.

21.11.6 Exam DSL source code

The Haskell source code for this project is in files:

- `ExamDSL_base.hs`, which is the skeleton to flesh out for a solution to this project
- `SimpleHTML.hs`, which is the module of HTML string templates

21.12 Acknowledgements

In Summer 2016, I adapted and revised much of this work from the following sources:

- Chapter 8 of my *Notes on Functional Programming with Haskell* [8], which is influenced by Bird and Wadler [1], Hudak [18], Wentworth [28], and likely other sources.

- My *Functional Data Structures (Scala)* [9], which is based, in part, on Chapter 3 of the book *Functional Programming in Scala* [5] and associated resources [6,7]

In 2017, I continued to develop this work as Chapter 8, Algebraic Data Types, of my 2017 Haskell-based programming languages textbook. I added discussion of the `Maybe` and `Either` types. For this work, I studied the Haskell `Data.Maybe` and `Data.Either` documentation, Chapter 4 on Error Handling in Chiusano [5], and articles on the Option Type [33] and the Null Object Pattern [26,32,34].

In Summer 2018, I revised this as Chapter 21, Algebraic Data Types, in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming* [12].

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on this textbook. In January 2022, I began refining the existing content, integrating additional separately developed materials, reformatting the document (e.g., using CSS), constructing a unified bibliography (e.g., using citeproc), and improving the build workflow and use of Pandoc.

In 2022, I added the descriptions of three projects: Carrie's Candy Bowl, Sandwich DSL, and Exam DSL. These are adapted from homework assignments I have given in the past.

I devised the first version of Carrie's Candy Bowl project for an exam question in the Lua-based, Fall 2013 CSci 658 (Software Language Engineering) class. It is algebraic data type reformulation of the Bag project I had assigned several times in CSci 555 (Functional Programming) since the mid-1990s. I revised the problem for use in the Scala-based CSci 555 class in 2016 and for later use in the Haskell-bsed CSci 450 (Organization of Programming Languages) in Fall 2018.

I devised the first version of the Sandwich DSL problem for an exam question in the Lua-based, Fall 2013 CSci 658 (Software Language Engineering) class. I subsequently developed a full Haskell-based project for the Fall 2014 CSci 450 (Organization of Programming Languages) class. I then converted the case study to use Scala for the Scala-based, Spring 2016 CSci 555 (Functional Programming). I revised Haskell-based version for the Fall 2017 CSci 450 class.

I developed the Exam DSL project description in Fall 2018 motivated by the Sandwich DSL project and a set of questions I gave on an exam.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

21.13 Terms and Concepts

Types, algebraic data types (composite), sum (tagged, disjoint union, variant, enumerated), product (tuple, record), arity, nullary, recursive types, algebraic data types versus abstract data types, syntax, semantics, pattern matching, null reference, safe error handling, `Maybe` and `Either` “option” types, Null Object design pattern, association list (map, dictionary), key, value.

21.14 References

[1] Richard Bird and Philip Wadler. 1988. *Introduction to functional programming* (First ed.). Prentice Hall, Englewood Cliffs, New Jersey, USA.

[2] Edwin Brady. 2017. *Type-driven development with Idris.* Manning, Shelter Island, New York, USA.

[3] Edwin Brady. 2022. Idris: A language for type-driven development. Retrieved from https://www.idris-lang.org

[4] Rod M. Burstall, David B. MacQueen, and Donald T Sannella. 1980. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on Lisp and functional programming*, 136–143.

[5] Paul Chiusano and Runar Bjarnason. 2015. *Functional programming in Scala* (First ed.). Manning, Shelter Island, New York, USA.

[6] Paul Chiusano and Runar Bjarnason. 2022. FP in Scala exercises, hints, and answers. Retrieved from https://github.com/fpinscala/fpinscala

[7] Paul Chiusano and Runar Bjarnason. 2022. FP in Scala community guide and chapter notes. Retrieved from https://github.com/fpinscala/fpinscala/wiki

[8] H. Conrad Cunningham. 2014. *Notes on functional programming with Haskell.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/Notes_FP_Haskell/Notes_on_Functional_Programming_with_Haskell.pdf

[9] H. Conrad Cunningham. 2019. *Functional data structures (Scala).* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ScalaFP/FPS03/FunctionalDS.html

[10] H. Conrad Cunningham. 2019. *Handling errors without exceptions (Scala).* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ScalaFP/FPS04/ErrorHandling.html

[11] H. Conrad Cunningham. 2022. *Notes on domain-specific languages.* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/DSLs/NotesDSLs.html

[12] H. Conrad Cunningham. 2022. *Exploring programming languages with interpreters and functional programming (ELIFP).* University of Mississippi, Department of Computer and Information Science, University, Mississippi, USA. Retrieved from https://john.cs.olemiss.edu/~hcc/docs/ELIFP/ELIFP.pdf

[13] Evan Czaplicki. 2022. Elm: A delightful language for reliable web applications. Retrieved from https://elm-lang.org

[14] Nell Dale and Henry M. Walker. 1996. *Abstract data types: Specifications, implementations, and applications.* D. C. Heath, Lexington, Massachusetts, USA.

[15] Richard Feldman. 2020. *Elm in action.* Manning, Shelter Island, New York, USA.

[16] Phil Freeman. 2017. *Purescript by example: Functional programming for the web.* Leanpub, Victoria, British Columbia, Canada. Retrieved from https://book.purescript.org/

[17] Tony Hoare. 2009. Null references: The billion dollar mistake (presentation). Retrieved from https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare

[18] Paul Hudak and Joseph H. Fasel. 1992. A gentle introduction to Haskell. *ACM SIGPLAN Notices* 27, 5 (May 1992), 1–52.

[19] Steve Klabnik, Carol Nichols, and Conributers. 2019. *The Rust programming language* (Rust 2018th ed.). No Starch Press, San Francisco, California, USA. Retrieved from https://doc.rust-lang.org/book/

[20] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in Scala* (First ed.). Artima, Inc., Walnut Creek, California, USA.

[21] purescript.org. 2022. PureScript: A strongly-typed functional programming language that compiles to javascript. Retrieved from https://www.purescript.org/

[22] Python Software Foundation. 2022. Python. Retrieved from https://www.python.org/

[23] Luciano Ramalho. 2013. *Fluent Python: Clear, concise, and effective programming.* O'Reilly Media, Sebastopol, California, USA.

[24] Rust Team. 2022. Rust: A language empowering everyone to build reliable and efficient software. Retrieved from https://www.rust-lang.org/

[25] Scala Language Organization. 2022. The Scala programming language. Retrieved from https://www.scala-lang.org/

[26] Source Making. 2022. Null object design pattern. Retrieved from https://sourcemaking.com/design_patterns/null_object

[27] Simon Thompson. 2011. *Haskell: The craft of programming* (Third ed.). Addison-Wesley, Boston, Massachusetts, USA.

[28] E. Peter Wentworth. 1990. *Introduction to functional programming using RUFL.* Rhodes University, Department of Computer Science, Grahamstown, South Africa.

[29] Wikpedia: The Free Encyclopedia. 2022. Abstract data type. Retrieved from https://en.wikipedia.org/wiki/Abstract_data_type

[30] Wikpedia: The Free Encyclopedia. 2022. Algebraic data type. Retrieved from https://en.wikipedia.org/wiki/Algebraic_data_type

[31] Wikpedia: The Free Encyclopedia. 2022. Nullable type. Retrieved from https://en.wikipedia.org/wiki/Nullable_type

[32] Wikpedia: The Free Encyclopedia. 2022. Null object pattern. Retrieved from https://en.wikipedia.org/wiki/Null_object_pattern

[33] Wikpedia: The Free Encyclopedia. 2022. Option type. Retrieved from https://en.wikipedia.org/wiki/Option_type

[34] Bobby Woolf. 1997. Null object. In *Pattern languages of program design 3*, Robert Martin, Dirk Riehle and Frank Buschmann (eds.). Addison-Wesley, Boston, Massachusetts, USA, 5–18.