CSE250 Week 4: ADTs and Performance Measures

But first: the idea of **Matchable Structure** (sprinkled in chs. 2-4 and 15)

This concept is best explained operationally, IMHO:

- Constants are matchable, variables are not.
- Variables inside other matchable structures can be declared and assigned in the act of a match.
- Tuples have matchable structure.
- Lists have matchable structure, but arrays do not (at least not directly).
- `None` and `Some(x)` are matchable structure for whether a value exists. If so, the value is assigned to `x`.
- `Nil` and `x :: rest` are matchable structure for whether a `List` is empty or not. In the nonempty case, the first element is assigned to x and the rest of the list to rest.
- Strings do not have directly matchable structure, but you can use their `headOption` method to return `None` from the empty string and `Some(c)` otherwise, where `c` is the first char in the string---and the `tail` method gives the rest of the string.
- Case classes provide matchable structure for their arguments.
- But inheritance hierarchies do not---this is not like Java `instanceof` or Python `isinstance` or C++ `dynamic_cast`. Not even when the arguments are `val`.
- A variable by itself matches anything; underscore `_` by itself is the way of saying the default case when you don't care about the value.

Matchable structure is used in the `match` .. `case` .. `=>` .. expression, which replaces the `switch` statement in C/C++/Java or JavaScript. Python 3.10 (2021) just adopted `match` .. `case`. Here are some examples (file `CaseExamples.scala` in the `ScalaSamples` folder):

Matching on constants (classic switch statement):

```
def isVowel(c: Char) = c match {
   case 'a' => true
   case 'e' => true
   case 'i' => true
   case 'o' => true
   case 'u' => true
   case _ => false
}
```

The cases aree sequential but there is no fall-through of the bodies (in C/C++/Java terms, it's as if each body ends with an automatic `break`). Cases treated the same way can be grouped with | on the same line---which might help you write methods that take care of more cases:

```
def isVowel2(c: Char) = c match {
   case 'a' | 'A' | 'e' | 'E' | 'i' | 'I' | 'o' | 'O' | 'u' | 'U' | 'y' | 'Y' => true
   case _ => false
}
```

Matching on the `Option[A]` parametric case-class (here with type `A` = `Char`):

```
def vowelsIn(str: String): List[Boolean] = str.headOption match {
   case None => Nil
   case Some(c) => isVowel2(c) :: vowelsIn(str.tail)
}

println("Which chars in \"AnyRef\" are vowels? " + vowelsIn("AnyRef"))
```

Note that this method produced a list but did not match *on* the list. Here are ones that do:

Matching on Lists

The `cubeRoots` example from Week 2 was a simple one-pass recursion over a list `cubesL`. It could have been written as a mapping via `cubesL.map(x = > scala.math.pow(x,1/3.0))` since it just works on each element separately. More substantial is when the list elements are compounded into a final total. These examples are in `ListSums.scala`:

```
def sumList(lis: List[Int]): Int = lis match {
   case Nil => 0
   case x :: xs => x + sumList(xs)
}
```

This produces just a single number, the sum. Suppose we want all the partial sums. Here is an example of a list and its corresponding list of partial sums:

```
List(1, 3, 6, 2, 5, 4, 2, 1, 9, 6, 4, 8)
List(1, 4,10,12,17,21,23,24,33,39,43,51)
```

Now we want to update the partial sum with each element from the list. We could use an external variable for that (a "global") but if we want to encapsulate the partial sums within the method, we can make it a second parameter that gets passed. The basic idea is called **Accumulator Passing Style**. (I have not found it called as-such in our text, but the term was used and briefly taught in CSE116 recently.)

```
/** List of prefix sums of an integer list, beginning with the first element
    Initial call should have psum=0.
 */
```

```
   def prefSums(lis: List[Int], psum: Int): List[Int] = lis match {
      case Nil => Nil
      case x :: xs => (x+psum) :: prefSums(xs, psum+x)
   }  //Recursion Invariant: sumList(lis) = psum+x+sumList(xs)
```

Note that the accumulator parameter `psum` is increased by `x` in the recursive call. This preserves the **invariant** that `psum` always equals the sum of the elements that have been seen so far. To establish this invariant, the initial call needs to be with `psum` = 0. (Or, if you really want to add the list sum onto another value, amke the initial call with that value.)

Here is a case of using two accumulator parameters:

```
/** List all maximal sequences of decreasing-length words in a list of strings.
    Using accumulators for both the phrase and the length enables the simplest kind of match.
    Because first word always begins a new chain, we want to begin with a super-high length #
 */
   def decreasingPhrases(words: List[String], phrase: String, lastWordLen: Int): List[String]
   = words match {
      case Nil => if (phrase == "") Nil else phrase::Nil
      case wd :: rest => if (wd.length >= lastWordLen) {   //decreasing phrase stopped
         phrase :: decreasingPhrases(rest, wd, wd.length)  //wd may begin new phrase
      } else if (phrase == "") {                           //skipping this test would leave
         decreasingPhrases(rest, wd, wd.length)            //an extra space out front
      } else {                                             //because of the recursion below.
         decreasingPhrases(rest, phrase + " " + wd, wd.length)
      }
   }

   val line = "The quick brown fox jumped over the lazy dog's back"
   val decreasingPhraseList = decreasingPhrases(line.split(" ").toList, "", 9999)
```

This yields `"The" :: "quick" :: "brown fox" :: "jumped over the" :: "lazy" :: "dog's back" :: Nil .`

Matching on Tuples

A match on a fixed-size tuple is just a combination of matches on the individual components. Here is a simple-minded example that uses matching to treat cases of zero specially (back to `CaseExamples.scala` for this one):

```
   def numberType(tup: (Double,Double)): String = tup match {
      case (0.0, 0.0) => "the origin"
      case (_,   0.0) => "nonzero real number"
```

```
      case (0.0, _)   => "nonzero pure imaginary number"
      case (_, _)     => "mixed complex number"
   }
```

Here the underscore is a "throwaway widlcard"---its value is not retrievable, and in the last case could be a clash of different values anyway. We could always get the values by `tup._1` and `tup._2` anyway, so the use of the lone underscore `_` is a way of saying "don't-care." [Remark: It is problematic to match on the floating-point zero that has been *computed*---as we saw with the "`...E-16`" values in the `cubeRoots` example---but matching on a *constructed* `0.0` field (especially when it is a `val` constructor argument kept constant) is generally OK.]

Now for an example that combines matching on tuples, lists, and on an accumulator parameter that is actively involved in the match. An accumulator parameter can embody a notion of "State", as in the "State Design Pattern." A recent CSE116 had an example with "**Hulk**" being in a state of calm or "angry" (the latter becoming a state of being big and green, too). On assignment 2, you can use the idea of Hulk being "alpha"---i.e., in the mode of building up alphanumeric characters---and stopping only when a non-alpha character is reached. In the following similar numerical example, the hulking up of an integer sum gets stopped when the sum goes over a given threshold, and the process starts again from zero ("Bruce Banner").

```
def thresholdSums(ell: List[Int], threshold: Int, psum: Int): List[Int] = (ell,psum) match {
   case (Nil,0) => Nil         // no more elements to add
   case (Nil,p) => p :: Nil    // psum :: Nil would be equivalent
   case (x :: xs, p) => if (x+p >= threshold) {
      (x+p) :: thresholdSums(xs, threshold, 0)
   } else {
      thresholdSums(xs, threshold, p+x)    // INVARIANT: psum < threshold in any call
   }
}
```

Here is a trace of a run of this code on List(1, 3, 6, 2, 5, 4, 2, 1, 9, 6, 4, 8) with threshold 10:

```
thresholdSums(1::3::6::2::5::4::2::1::9::6::4::8::Nil,  10,  0)
thresholdSums(3::6::2::5::4::2::1::9::6::4::8::Nil,  10,  1)
thresholdSums(6::2::5::4::2::1::9::6::4::8::Nil,  10,  4)     since 6+4 ⩾ 10
10 :: thresholdSums(2::5::4::2::1::9::6::4::8::Nil,  10,  0)
10 :: thresholdSums(5::4::2::1::9::6::4::8::Nil,  10,  2)
10 :: thresholdSums(4::2::1::9::6::4::8::Nil,  10,  7)        since 4+7 ⩾ 10
10 :: 11 :: thresholdSums(2::1::9::6::4::8::Nil,  10,  0)
10 :: 11 :: thresholdSums(1::9::6::4::8::Nil,  10,  2)
10 :: 11 :: thresholdSums(9::6::4::8::Nil,  10,  3)           since 9+3 ⩾ 10
10 :: 11 :: 12 :: thresholdSums(6::4::8::Nil,  10,  0)
10 :: 11 :: 12 :: thresholdSums(4::8::Nil,  10,  6)           since 6+4 ⩾ 10
```

```
10 :: 11 :: 12 :: 10 :: thresholdSums(8::Nil,  10,  0)
10 :: 11 :: 12 :: 10 :: thresholdSums(Nil,  10,  8)          now in a base case
10 :: 11 :: 12 :: 10 :: 8 :: Nil
```

So we got `List(10, 11, 12, 10, 8)` as the final answer. If there hasd not been a final 8, we would have ended with the base case `thresholdSums(Nil, 10, 0)`. Then we don't want to attach a useless 0 onto the list, so we use the first case of the `match` to make that not happen.

Matching on Case Classes

The match on `None` and `Some(c)` already comes from a special **case class**. The simplest kind of this is how Scala implements "enum"s. It is legal for case classes to inherit from a *trait*, and the text's example on page 136 is improved by making it a `sealed trait`:

```
sealed trait StreetLightColor

case object Red extends StreetLightColor     //case object since there is only one "Red"
case object Yellow extends StreetLightColor
case object Green extends StreetLightColor

class StreetLight(private var _color: StreetLightColor) {
   def color = _color

   def cycle: Unit = _color match {   //this is a mutator procedure
      case Green => _color = Yellow   //here Yellow is a zero-parameter constructor call
      case Yellow => _color = Red     //that constructs an object of type StreetLightColor
      case Red => _color = Green
   }
}

object StreetLight extends App {
   val light = new StreetLight(Red)
   for (i <- 0 until 4) {   //4 is exclusive
      light.cycle
   }
   println("After 4 cycles starting from Red, the light ended up " + light.color)
}
```

Note that because there are no parameters to match, not like `c` in `Some(c)`, there is only one possible object of each color, `Red`, `Yellow`, or `Green`. Hence their declarations say `case object` not `case class`, and then they can have no `constructor arguments` nor parentheses `()` at all. Saying `sealed` tells the Scala compiler that there are no more possible `StreetLightColor` options other than those in this one file. This allows the compiler to avoid having to "cycle" through the match cases

when executing a call to `cycle`---instead it can create a direct **jump table** for whichever color the light is at the moment.

The one bad thing about this code structure is that the names `Red`, `Yellow`, and `Green` are dumped into the top-level namespace. They could clash with the same names being used in other enumerations, such as `Rainbow`. This issue happens with `enum`s in the C/C++/Java world too, which is a reason why C++ has `namespace` and Java leans more on packages. Coding the colors as fixed integer fields of a class `StreetLightColor` (as the text shows first) would disable the advantage of matchable structure. That is why Scala provides a way to have the name protection and use the same match syntax:

```
object StreetLightColor extends Enumeration {
   val Red, Yellow, Green = Value      //order doesn't matter; Value is special syntax.
}
```

The only difference to the client is that the color type is now `StreetLightColor.Value`, which is an example of an **inner class** in Scala. The client code needs an extra statement `import StreetLightColor._` to use the color-class names without the prefix `StreetLightColor.` But outside the client, the prefix prevents the "global namespace" from being "polluted."

ADTs (now in Chapter 7 but harking back to Chapter 6)

An **Abstract Data Type** (**ADT**) is more than just a "data object" or some objects combined into a "data structure". It also specifies:

1. The operations provided to clients (end-clients and/or library coders) for building and manipulating the objects;
2. Logical requirements and guaranteed properties of those operations; and
3. Efficiency specifications (at least relative) for those operations.

The text only mentions point 1 when it defines ADTs in its short section 7.1, but it addresses point 3 right away, and it covers point 2 first at the level of **unit testing** of individual pieces of code (section 7.7). I tend to approach point 2 first at the level of the whole ADT analytically, per what is called design by contract. Here is a small example that is reflected accurately by the text's little box on "Comparators" on page 143 in chapter 4. Scala inherits from Java an ADT called `Comparator[T]` that requires the parametric type `T` to implement < and = comparisons. It provides an operation `x.compareTo(y)` that is basically always implemented to have the results

$$x.compareTo(y) = \begin{cases} +1 & if\ x > y \\ 0 & if\ x = y \\ -1 & if\ x < y \end{cases}$$

However, the **contract** only guarantees that the value is a positive integer if $x > y$ and a negative integer if $x < y$, not the specific integers $+1$ and -1. You may be able to see this, including the Java `@Contract` attribute, if you write a line of code with (say) `true.compareTo(false)` and hover over the `compareTo` (or right-click it?) in your IDE. It is common---but risky---to use `bexp.compareTo(false)` as a way of converting a Boolean expression to have value 0 when false and $+1$ if true.

The one thing that an ADT is not supposed to specify is *how* the data type is implemented. That is the "Abstract" aspect. The library designer must be free to change the implementation, so long as nothing in the API or contract or complexity specs changes. (All three can be enhanced.) Achieving certain promised combinations of logic and performance for a given API, however, often dictates much of the implementation. *That correspondence is the main subject of this course.*

Set and Map as ADTs

The three most basic operations provided by all kinds of `Set` objects are *creating* sets by various means, *quick lookup* of individual objects (i.e., testing whether a given object is in the set) and *traversing* through all objects in the set. A Mutable Set adds operations to modify a Set after it is created by adding and/or removing elements. It may---or may not---share implementation details with the basic (immutable) Set. One key logical property in the contract for a Set is:

- A `Set` never has two copies of the same value. This may `require` the argument type `A` (which I will also call the "client type" or the "foreground type") to implement `==` as value equality or otherwise have an `equals` comparison, so that the logical requirement "x is not the same as y" for any two elements x,y in the Set is implemented as `!x.equals(y)`.

Here are some further properties of interest:

- If the foreground type `A` in `Set[A]` has `<` as well as `==`, then is a traversal guaranteed to be in a totally sorted order?
- If not, can we at least say that if a `Set` object `foo` is created *the same way* by two different lines of code (or the same line at two different times in the execution, or by two different threads at the same time), which then traverse the set, will the two transversals give the same sequence of elements?

A **Map** is actually just a more useful kind of Set that returns a value when you successfully look up an element. So `Map[A,B]` gives values of type `B` when you find an element of type `A`. The key logical property of a Map is that the same element always gives the same value.

Time Complexity and O-Notation

[This will come during Monday's lecture, using the materials given on Assignment 2.]