1. My first instinct is to take advantage of how I'm going to store the data (described in question 2). Start parsing character by character. When I reach a USFM flag, pass control over to that object class. For example, if I come across a \c flag with a 4 after it, I make a chapter object containing chapter 4. It then keeps reading and finds a \v flag. This then makes a verse object and continues parsing. The verse object will then be stored in the chapter.
2. I would keep the verses stored in objects. From the example I'd need 4 object classes: an ID class, a Heading class, a Chapter class, and a Verse class. After parsing the text as described in question 1, I'd have a tree like structure where the ID has a name and a list of Headings (or books). Each Heading has a list of Chapters. And each Chapter has a list of Verses. Instead of lists, maps would probably be best as you would allow for some, but not all verses or chapters to be present.

```
Public Class ID {
    String name;
    Map<String, Heading> headings;
}
Public Class Heading {
    String name;
    Map<Int, Chapter> chapters;
}
Public Class Chapter {
    Int name;
    Map<Int, Verse> verses;
}
Public Class Verse {
    Int name;
    String verse;
}
```

3. Because each verse can be referenced and printed by the path to it (book, chapter, verse), and each one has a unique identifier, you can have two ID objects next to each other: one untranslated and one translated. The map allows for the translated version to not have everything that the untranslated does (or vice versa if that'll ever happen). I picture a display with a few possible views.
   One shows the untranslated text and if you click on a verse, a box appears with the translated text (there can also be a different color highlight if the verse has or hasn't been translated).
   Another can show the translated text. But if there is a verse in the untranslated text that doesn't show up in the translated text, it would appear as a muted color or italicized to indicate that it hasn't been translated.
   In either case, by clicking on a verse, you can edit/create the translated version.
4. Well my whole assumption was that verses were neatly contained in chapters and so on. So the fact that red letters might not be neatly contained in verses kind of ruins the whole structure. My new approach would have to be something about keeping track of markers that point to the text. I could have a binary tree that allows you to store the location of markers while also allowing you to lookup the last marker given a location. This would allow you to keep the user interface because when the user clicks on the verse, it can easily look up what verse it is a part of and then where it is in the translated version. You could also colorize the red letters easily by just checking if the last r marker was the beginning or end of the red letters.
   Some problems with this approach are that if you add or edit a verse, you have to update every marker following, even as far as into other books. This can be gotten around by making checkpoints perhaps at the book or chapter level which are independent of each other.