CREATING HTML CONTENT THAT CONTAINS HTML ELEMENTS

UMD CS DEPARTMENT

Abstract. On occasion, we will provide a focused document entitled "Class notes" when we wish to draw students' attention to some subtle topics that are discussed in class but might require more time and focus than in available in a lecture setting.

In this, the first of several class notes, we discuss how HTML may be instructed to parse its data as pure text in some settings and to "evaluate" it or render it as mark-up in other, more common settings. In order to do this, however, we have to spend some time discussing the structural properties of whole numbers so that we can fluently translate them between at least two bases that are important in CS applications, base-2 and base-16. We note, in closing, that these are not merely academic considerations: HTML and CSS both often express color codes in hexadecimal rather than decimal notation.

1. The general problem

Suppose that I am trying to create an HTML document that displays the following text:

> Some tags, such as `<h1>`, and `<p>` clearly work with horizontal layout, whereas the list elements, `<ul>`, `<ol>`, and `<dl>`, determine vertical positioning.

To do this I must instruct HTML to determine when it is to ignore HTML elements and treat them as any other kind of text (data), but at other times to interpret the HTML elements as commands. For example, consider the following HTML:

```
<p>This is <strong>important</strong> data.</p>
```

If we give this to any browser as written we should get something that looks like:

This is *important* data.

The problem we're trying to solve is to have the Browser render the data as it originally appeared, i.e, uninterpreted:

```
<p>This is <strong>important</strong> data.</p>
```

In other words, we need some way to tell HTML to "quote" text in some instances but to evaluate it, i.e., interpret it as another HTML command, in other instances. If this sounds tricky, it is.

1.1. **A little Computer Science helps here . . .** Earlier, we talked about *reductionism*, a way of thinking where complex systems or behaviors are the result of composing simpler, atomic elements. For example: sub-atomic particulars combine to make atoms, atoms combine to make molecules, etc. The idea is that complex structure can be decomposed into a finite collection of "atomic" entities.

In current computing models, all computing behaviors (systems) can be reduced to binary units, i.e., "bits," which we can think of as "off" or "on" states, or "false" and "true" or any "binary set," for that matter. Because we are interested in designing and producing web-sites, we are focusing on a subset of computing behaviors and it is helpful, at this time, to recast what we already know in terms of these "bits."

1.2. **HTML is text-based.** Text is comprised of alphabetic characters, punctuation, numerals, and "special symbols," such as the kinds of symbols we see in mathematics, music, many card games, etc. Recall that the preamble for our HTML documents has certain requirements:

```
<!DOCTYPE html>
<html lang="en">
  <head>
     <title>Text that appears as the name of this page.</title>
     <meta charset="utf-8">
  </head>
  <body>

  </body>
</html>
```

The `charset` attribution for the `meta` element is *not automatically provided* for you by the KOMODO editor; you need to physically add it. If you do not choose to add a `charset` binding, the Validator will complain and some "default" character set will be loaded.

1.2.1. *Charsets.* A "charset" is a dictionary-like structure that "maps" symbols to numeric values which are ultimately represented by the machine as "bit" strings. The "standard" alphabet that appears in Western languages is often represented in ASCII code, which is a mapping from alphameric characters to binary (base-2) numbers in the range from 0 to 127, which is conveniently represented in base-2 arithmetic as binary digits from 0000000 to 1111111. You should access a binary converter/calculator page, for a set of tools to help you visualize binary numbers and their decimal (base-10) equivalents.

Obviously, interesting web pages require more than these 128 characters. If you navigate to some images of ASCII tables you should be able to access the extended ASCII tables for HTML. You can read through this table and find the ASCII symbols for the < and > symbols that delimit HTML tags. Using those translations, you should now be able to have HTML format any text that wishes to embed HTML elements as data, i.e., as textual elements, rather than "executing" them as commands. Try to format:

```
<p>Some horizontal elements include <h1>, <h2>, through <h6>,
and some vertical elements include the three basic list types:
<ol>, <ul>, and <dl>.</p>
```

1.2.2. *Using the explicit values of ASCII characters in the HTML character stream.* Let's take a closer look at one way you might choose to represent the text above. The ASCII character set is a "default" for Western languages, and so it is certainly used by your browser. According to the table, the symbol < may be obtained as the less-than operator in mathematics, and similarly one may use the greater-than symbol, >. Note that the "values" for these symbols are the decimal (base-10) numbers 60 and 62, and that these numbers are given in two other bases —base-8, octal and base-16, hex or hexadecimal.

Knowing this, you should now see that the explicit usage, as `&#60;` for < and `&#62;` for >, works because the HTML parser treats objects that begin with ampersands as "special"—in fact, they are called "escape" characters.

By the way, you could also write `&#x3c;` for < and `&#x3e;` for >. In these usages, the `x` immediately preceding each number informs the parser that we are using hexadecimal rather than decimal representations. And, while we are on the subject, a "purist" might object that we're using mathematical symbols where textual symbols are intended. This may not have any immediate impact on what gets rendered in the browser, but it might if a particular browser spaces mathematical symbols differently than textual ones. We could (and should) use the UTF-8 charset escape sequence for < and > as "braces" as opposed to mathematical relations. Should you look that up, you'll find that these are bigger decimal numbers than their ASCII counterparts, and one might indeed prefer the more compact base-16 encoding in everyday usage; see if you can find these for yourselves.

2. Refining our understanding of numbers

We have been tacitly using terms such as "base-10" and "base-2" to describe ways of representing numbers. (Actually, we have been careful to talk only about positive whole numbers.) Numbers are *structures* —objects that are constructed, and that have specific properties. A little later in this course, we will devise programs to translate between several commonly used bases in computing: base-2 (binary), base-10 (decimal), and base-16 (hexadecimal).

2.1. **Constructing and deconstructing base-10 numbers.** Consider the number 328_{10}. If you remember your basic arithmetic, this number can be expressed using the distributive property:

$$328_{10} = (3 \times 10^2) + (2 \times 10^1) + (8 \times 10^0).$$

(Recall that any number raised to the power of 0 is equal to 1, so that last term is just $8 \times 1 = 8$.)

Suppose that we wish to express this base-10 number as its equivalent in base-2 arithmetic. Ask: what is the nearest power of 2 to 328? Well, $2^8 = 256$ and $2^9 = 512$, so 256 is closest, and the difference of $328 - 256 = 72$. Ask now, what is the nearest power of 2

to 72? Well, $2^6 = 64$ and the difference $72 - 64 = 8$, and $2^3 = 8$. List out the powers of 2 that we needed to reduce the original base-10 number:

$$(1 \times 2^8) + (1 \times 2^6) + (1 \times 2^3) = 256 + 64 + 8 = 328$$

Now, we cannot just "skip" the missing place values (they are place holders); put these missing placeholders in now and get:

$$(1 \times 2^8) + (0 \times 2^7) + (1 \times 10^6) + (0 \times 10^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)$$

And, in base-2 this is written: 101001000_2 in just the same way that we write:

$$(3 \times 10^2) + (2 \times 10^1) + (8 \times 10^0)$$

as 328_{10}.

2.2. **Compact representations.** Obviously, writing common numbers in base-2 requires lots of space. For this reason, base-16 (hexadecimal) is often used instead. To convert base-10 to base-16, we follow the same procedure as we used above. We need to augment our digits to account for the extra numbers from 10 to 15. Traditionally, this is done by using alphabetic characters a through f, so the digits in base-16 are:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f$$

It turns out, by the way, that we won't need the "extra" numerals to convert our original base-10 number to base-16, but let's do it anyway. Ask: what is the closest power of 16 to 328? $16^2 = 256$, and the difference $328 - 256 = 72$, and the next power of 16 is just 16 and $4 \times 16 = 64$, leaving $(72 - 64) = 8$, which is (8×16^0). Putting these together:

$$(1 \times 16^2) + (4 \times 16^1) + (8 \times 16^0)$$

which we write as 148_{16} (base-16).

2.3. **Why you will care.** Why do we care about being to work with different number systems? Fair question. Pure binary digit representations are rarely relevant in introductory classes, such as this one. Base-16 (hexadecimal), however, is particularly relevant to you because, as you will see, color are often coded in hexadecimal to conserve space.

3. Where do we go from here?

As I mentioned in the introduction to these class notes, we'll revisit these ideas later, after learning enough JavaScript to write our own logic to automate these conversions (and more).