“Smart” Short Answer or Fill-Blank Questions for Desire2Learn Brightspace

Bob Brown
College of Computing and Software Engineering
Kennesaw State University

Auto-grading in Desire2Learn Brightspace can take a load off the professor’s shoulders, or it can be a burden. It’s burden when a student gets a short answer or fill in the blank question right but because of fill words, articles, or punctuation the student’s answer fails auto-grading. Until recently Desire2Learn only allowed answers to be case-insensitive and allowed for multiple correct answers, so there was just too much to go wrong.

Consider the question, “How can one tell by inspection whether a two’s complement binary number is negative?” These are all correct answers:

The sign bit is one
the sign bit is 1.
The leftmost bit is 1.
the left bit is one
There is a one in the sign bit.

There are probably many others, but the key words are *sign* **or** *left(most)* **and** bit **and** *1* **or** *one*. The last example reverses the order of things but is equally correct. Even the presence or absence of punctuation defeats strict character-for-character matching, but if we can identify keywords and their order, we can write “smart” answers for Desire2Learn auto-grading.

Regular Expressions to the Rescue!

In 2020 Desire2Learn introduced the ability to use a *regular expression* as an answer to a short answer or fill-blank question. A regular expression is a pattern that is compared to a string of characters and might match all or part of the string. The professor writes the pattern and Desire2Learn checks the students’ answers against the pattern. Regular expressions are powerful, arcane, and even a little dangerous. If that makes you say, “Oh boy!” this document is for you. If it makes you say, “Oh no!” this document is still for you. It will help you unlock the mysteries and use the power of regular expressions. A little dangerous? That’s what testing is for, and we’ll cover that toward the end.

A regular expression pattern consists of two kinds of things, ordinary characters, which must match the target string exactly, and metacharacters, which have special meaning and can control the pattern matching process. The idea is similar to, but much more powerful than, the “match anything” wildcard character * with which you may already be familiar.

In what follows, regular expression patterns will be shown in bold and monospace or put on a line by themselves when necessary, so they don't get confused with sentence punctuation.

Let's look at a very simple example. We might want to allow either the American or British spelling of the word color/colour. Before regular expressions we'd have had to give two separate answers. Now we can just write the pattern **`colou?r`** All the letters are regular characters; they match themselves exactly. The **`?`** is a metacharacter, specifically a quantifier. It means zero or one and follows the pattern character or characters to be quantified. So, that pattern matches c-o-l-o, then zero or one letter u and finally the letter r. To make it case-insensitive we'd prefix the pattern with the metacharacters **`(?i)`**, so it would be **`(?i)colou?r`**. As you can see the question mark serves a different purpose when it's within parentheses.

What if we need a question mark in our pattern? The backward slash is the escape metacharacter. It removes the special meaning from the following character, so **`color\?`** would match exactly c-o-l-o-r-?

There's a complete guide to Brightspace regular expression metacharacters on line. Beware! At the time this was written, the sequence for case insensitivity was incorrect. It really is **`(?i)`** as given above. The example on that page is correct, it's only the first column that's in error.

Here are the regular expression metacharacters:

\ | ( ) [ ] { } ^ $ * + ? .

As you have seen, they're used in combination as well as alone. You've seen the use of the backward slash to escape metacharacters. The backward slash is also used "in reverse" to make letters that would otherwise be normal characters become metacharacters. For example, **`\b`** matches a word boundary instead of an ordinary letter b. More on that in a minute.

Regular Expression Classes

The power of regular expressions comes from classes and quantifiers. We saw **`?`** as a quantifier and we'll look at others in a moment.

A regular expression class is enclosed in square brackets and matches *one* of a set of characters. The class **`[abc]`** matches *one* of **a**, **b**, or **c**. The class **`[aeiou]`** matches any *one* character that's a vowel. The carat symbol inside square brackets inverts the class, so **`[^aeiou]`** matches any *one* character that's not a vowel.

The good news is that many of the most useful classes have been predefined for you.

`\w` matches one word character, namely *one* of **`[a-zA-Z0-9_]`**
`\W` matches *one* non-word character; in general, capitalization inverts the class.
`\d` matches *one* digit, 0, 1,2, 3, 4, 5, 6 7, 8, 9; **`\D`** matches *one* non-digit, *i.e.,* anything else.
`\s` matches *one* space character: space, tab, or new line; **\S** matches *one* non-space character.
`\b` matches a word boundary and is used for "whole word" matches. More in a moment.

. (a period) isn't really a class; it matches *any* single character. (To get a "real" period, escape it with a backslash: **\.**)

Quantifiers

The word *one* is italicized everywhere above to emphasize that the classes match exactly *one* character. What if we want more than one? Quantifiers! The quantifiers *follow* the expression being quantified.

? signifies one or none, as in the **colou?r** example above.
***** signifies zero or more
+ signifies one or more
{*count***}** specifies an exact count, so **xy{3}z** is the same as **xyyyz**

Word Boundaries

We said **\b** matches a word boundary to allow whole-word matches. Let's see some examples:

\bFred\b matches "Fred" but not "Frederick" or "AlFred" because there are word boundaries at the beginning and end.

\bFred.* matches "Fred" and "Freddie" and Frederick" and "Fred is a great guy" but still not "AlFred" because of the word boundary at the start. Remember that a period matches any character and * means zero or more of them.

(?i).*Fred\b matches (case-insensitive) "Fred" and "Alfred" and "AlFrEd" and "Oh, look, it's Fred." The period at the end of the last example is *not* matched, but D2L would still consider it a match because D2L counts any match *within* the string as a match.

Alternatives, Grouping and Anchors

In general, the parts of a regular expression are implicitly connected by *and*; that is, all parts must match. Sometimes you need to offer alternatives. In the question above, we want the student to type "sign" or a word starting with "left." The vertical bar **|** is the regular expression *or* (alternative) connector. Our partial regular expression would look like this:

```
\bsign\b|\bleft.*\b
```

We want to match the whole word "sign" *or* the string "left" that starts on a word boundary followed by zero or more (**.***) of any character, then a word boundary. The vertical bar near the center of the regular expression accomplishes that.

We aren't quite done yet. Often a sub-expression with an alternative is part of a larger regular expression. In this case, we want to add "bit." The expression above must be grouped so that we get one *or* the other of the terms in the group *and* the following term. Parentheses are the grouping operators:

```
(\bsign\b|\bleft.*\b).*\bbit\b
```

That does it! We match either the whole word "sign" *or* "left-something," then zero or more (**.***) *and* the whole word "bit."

The regular expression parser scans until it finds something that matches the first part of the pattern, so "The sign bit…" matches and the fill word "The" is ignored. Sometimes we want to insist that matching start with the first thing in the answer, or end with the last thing. The anchor metacharacters **`^`** and **`$`** accomplish that. The carat symbol anchors the front of the target string and the dollar symbol anchors the end. (The carat symbol also inverts a class definition, but only if it immediately follows a left bracket.)

So, the regular expression **`^pearl`** would match "pearls are" but not "my pearls" because the match is anchored to the beginning of the string.

Similarly, **`gold$`** anchors to the end of the target string and matches "I like gold" but not "The sunset is golden."

Some Real Examples

Let's look at some real examples; these are from the computing disciplines, but it should be easy to imagine them recast into your own discipline.

Question: How many bits are needed to represent the decimal number 1967 in binary?
Answer: **`(11|(?i)eleven)`**
You could record "11" and "eleven" in traditional fill-blank format, but what if the student typed, "It takes eleven bits"? That would fail in the traditional format but would be scored correctly using a regular expression. Even "eleventy" would get credit; you could prevent that by using **\b** word boundary metacharacters around "eleven."

Here's the rub: "It needs 11 or 12 bits," would also get full credit. I've chosen to ignore that, but if you have Smart Aleck students, you could do this:

```
(11|(?i)eleven)(\s+bits)?\.?$
```

This starts out with the original expression, then **`(\s+bits)?`** one or more space characters followed by the word "bits" zero or one times, **\.?** a period zero or one times, followed by **`$`** which anchors the end of the string; what the student typed must end with "11" or "eleven" followed by a space and "bits" and an optional period. "Twelve or eleven" would still get credit, and "11 binary digits" would fail. The latter is easy to fix with **`(bits|binary digits)?`** But the former is harder.

It's important to remember that this is not artificial intelligence, it's human intelligence expressed using a pattern-matching language. You will need to spot-check correct answers and, for each question, keep a list of test data.

Queston: What are the three steps, in order, of the instruction cycle of a von Neumann computer?
Answer: **`(?i)\bfetch\b.*\bdecode\b.*\bexecute\b\.?$`**

The three steps are fetch, decode, and execute. The regular expression requires all three words, optionally followed by a period (**\.?**) and anchored at the end of the string.

Question: How can you tell by inspection (just looking) whether a given two's complement binary number negative? (This is the question introduced at the beginning of this paper.)
Answer: **`(?i)(\bleft.*|\bsign\b).*\bbit\b.*( \b1\b|\bone\b)`**
`(?i)( \b1\b|\bone\b).*( \bleft.*|\bsign\b).*\bbit\b.*`

This one needed two regular expressions to account for "The sign bit is one," and "There's a one in the sign bit." Two regular expressions require two answers in D2L. We could have combined those with a vertical bar alternate metacharacter and more parentheses, but at some point, regular expressions get too dense even for the person who wrote them to decode them.

Answers with Partial Credit

D2L allows you to assign a weight to each answer, with 100% being full points, 50% being half off, etc. There's a trap here! D2L evaluates in order until it hits a match, then applies that weight. If you have multiple answers with different weights, you must list them in *descending order of specificity*.

For example, consider a question where "underflow" is the correct answer, but you want to give half credit for either "positive underflow" or "negative underflow." You must list the latter regular expression first, like this:

`(\bpositive\b|\bnegative\b)\s*\bunderflow\b`	50%
`\bunderflow\b`	100%

Why? Because if you listed **`\bunderflow\b`** first, either "positive underflow" or "negative underflow" would match and get full credit.

And also…

In the "Submission views" tab of your quiz or exam you must un-check "Show correct answers" if you have used regular expressions because D2L will show the regular expression as the correct answer. There's nothing secret about those regular expressions, but most students won't recognize them and that will cause unnecessary anxiety.

If you can't (shouldn't) show a regular expression as the correct answer, you must provide an explanation of the correct answer as a feedback item. Because "Show correct answers" applies to the entire quiz or exam, you'll have to provide feedback for all the questions. That's a Good Thing because it provides information at the very instant when the student is most engaged in learning that information.

The feedback need not be lengthy. Here's what I wrote for the "sign bit" question:

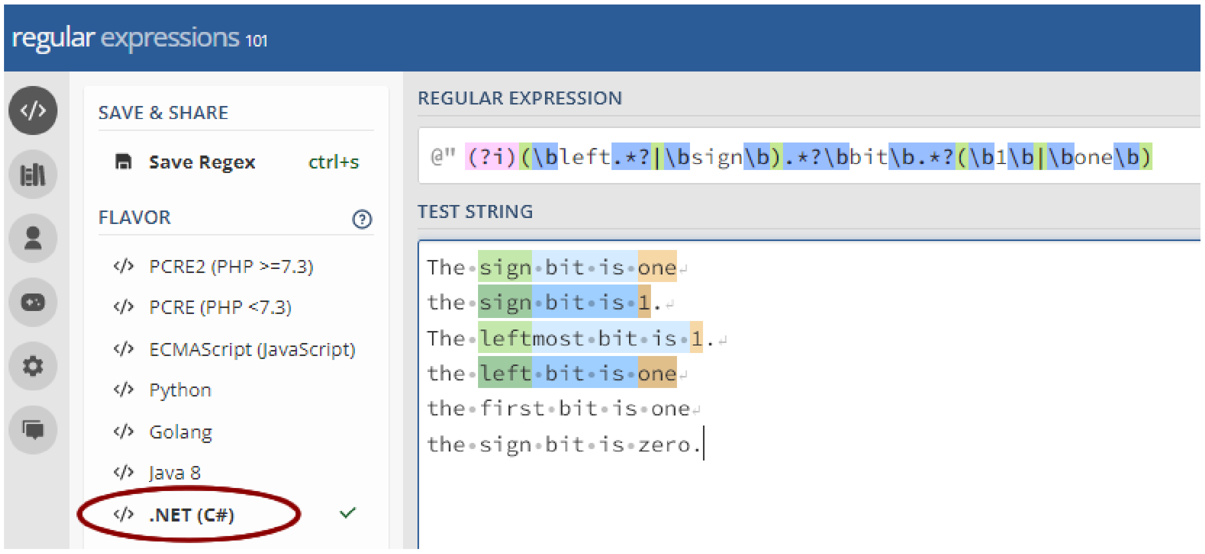
> *In a two's complement binary number, the leftmost bit is the sign bit. If the sign bit is a one, the number is negative. (Saying "first bit" or "last bit" is not meaningful when*

talking about binary numbers because whether one is counting from the left or right is determined by context.)

Testing Regular Expressions

Desire2Learn has not provided a good way to test regular expressions, and regular expressions are powerful, arcane, and even a little dangerous. You really do need to test. Happily, Brightspace Community member Craig Romanec has posted that D2L Brightspace uses Microsoft's .NET regular expression parser, and there's a compatible tester online. Be sure to select **.NET (C#)** under *FLAVOR* at the left. (If you use this a lot, consider using the Donate button to send a few bucks to help pay for server and bandwidth rental.)

Here is a part of the example given at the top of this paper:



FLAVOR: [✓ .NET (C#)]

REGULAR EXPRESSION

```
@" (?i)(\bleft.*?|\bsign\b).*?\bbit\b.*?(\b1\b|\bone\b)
```

TEST STRING

```
The sign bit is one
the sign bit is 1.
The leftmost bit is 1.
the left bit is one
the first bit is one
the sign bit is zero.
```

The regular expression is at the top, **.NET (C#)** is selected at the lower left, and several potential answers are typed into the test string box. D2L would count any of the highlighted strings as correct even if the entire string is not highlighted. The last two test strings are incorrect. The last one contains "sign bit" but does not contain *1* **or** *one* and the pattern requires it. The *entire* pattern must be satisfied for a match to be recognized by D2L, but characters not in the pattern may be present. If that's bad, you can use the anchor metacharacters **^** and **$** to limit what gets ignored.

Testing is important even after you've become very confident with regular expressions. Remember, they're powerful, arcane, and even a little dangerous!

When Students Complain

Expect your regular expressions to miss some plausibly correct answers and maybe even accept some incorrect answers. Your students will alert you in the first case. Spot-checking helps with the latter case. Do not despair! Just add to your test cases and tighten up your regular expressions. Just be sure small errors in your regular expressions do not disadvantage your students.