Guide to Regular Expressions

Regular expressions are an amazingly powerful tool for describing collections of strings. However, they can take some time to get used to and represent a totally different problem-solving strategy than the techniques for designing automata. This guide outlines several common strategies for designing regular expressions, showing how to use them by working through two examples in depth.

Quick Review: Compound Regular Expressions

Fundamentally, regular expressions describe patterns by combining individual strings together in three different ways:

- Union represents ***choice***. Intuitively, you can think of a regex of the form $A \cup B$ as saying "something matching pattern A or something matching pattern B."
- Concatenation represents ***sequencing***. You can think of a regex of the form AB as meaning "match pattern A, then match pattern B."
- Kleene stars represent ***iteration***. You can think of a regex of the form \(R^\star\) as saying "as many copies of R as you'd like."

Learning to write regexes, in a sense, means internalizing these tools and figuring out how best to apply them. It's good to keep these intuitions in mind as you read through what follows.

Write Out Concrete Test Cases

Often times, you'll need to design a regex that matches all strings meeting some criteria. Those criteria can be expressed in many different ways. Before you even start writing your regex, take a few minutes to write out two groups of strings: one group of strings that does meet the criteria, and one group that doesn't. The positive examples will help you hypothesize patterns that you can use as the basis for your regex. The negative examples will help you test those hypothesized patterns. And both groups will serve as useful test cases once you have a regex.

Let's see this in action. Consider this language L:

$$L = \{ w \in \{\texttt{a}, \texttt{b}\}^\star \mid w \text{ begins and ends with the same letter } \}$$

We're going to write a regular expression for this language. To do so, we'll begin not by guessing at the shape of the regex, but rather by simply listing a bunch of strings in and out of the language. Here's one such list:

Strings in L	Strings Not in L
`abba`	`aabb`
`aa`	`ab`
`a`	ε
`bbb`	`abaaaab`
`babab`	`baba`
`b`	`baaaaaa`
`baaaaaaab`	`aaaaab`
`abbaba`	`abaaababbabb`
`bb`	`babababa`
`aaaaaaaa`	`abbb`

In writing this list, I aimed to get as much variety in the strings as I could. I looked for very short strings and for strings of more modest length. I looked for strings consisting of lots of copies of one character and for strings using a mix of characters. Some strings are formed by repeating patterns, while other strings have characters that are more or less random.

The purpose of doing all this is to get stuff out there so that we can look and see if there

are any patterns. Plus, thinking about unusual cases this way makes it less likely that we're going to miss an edge case. In particular, the fact that I've written out strings of lengths 0, 1, and 2 will turn out to be very important!

Now that we've written out these strings, we can start looking for some patterns, beyond the obvious "all the strings in the language start and end with the same character." (That much we already knew from the language description.) Here are a few things we can note:

- The empty string isn't in the language, so every string in L has to have length one or more.
- Strings consisting of just a single character are in L. Stated differently, if a string has length 1, then "the first character" and "the last character" are always the same because there's only one character.
- If the string has length two or more, then what comes between the first and last characters of the string are irrelevant.

All of these observations are things that, in principle, we could have made purely by seeing the definition of L, but which are much easier to come by with concrete strings written out in front of us.

Now that we have these observations, the question is what we do with them. That's the topic for the next major piece of advice we have for designing regexes.

Encode Distinct Choices Using Unions

If we take a look at the strings we've written out above that are in the language, we can see that they generally fall into one of three categories:

1. Strings that are one letter long.
2. Strings that begin with an `a` and end with a second copy of `a`.
3. Strings that begin with a `b` and end with a second copy of `b`.

Each of these three cases is much simpler than the original, broader language L. In fact, we're now at a point where we can start writing regexes for each individual case.

1. For strings of length one, we have many options. One would be to simply list all the options for a one character string and write $\mathtt{a} \cup \mathtt{b}$. But a more compact way to do this would be to just use Σ, which is a regex meaning "any individual character."
2. For strings that begin with `a` and end with a second copy of `a`, we can write $\mathtt{a}\Sigma^\star\mathtt{a}$. This says "match an `a`, then anything (that's what $\Sigma^\star$ means), then anothe `a`).
3. Analogously, we can match strings that are bookended by two `b`'s as $\mathtt{b}\Sigma^\star\mathtt{b}$.

We've built individual regexes for each of the three cases - and that means that the finish line is in sight! All we have to do now is write a regex that says "anything matching any of these cases." That's where the $\cup$ (union) operator comes in. By simplying unioning all these together, we get a regex that covers all of the needed cases:

$$\Sigma \cup \mathtt{a}\Sigma^\star\mathtt{a} \cup \mathtt{b}\Sigma^\star\mathtt{b}$$

You might be wondering whether there's a way to combine those last two cases together - do we really need to list `a` and `b` separately? Alas, the answer is "no, there's no way to combine those." Regexes don't have a notion of "memory" in the sense of "remember the first character you read." (That being said, some regex libraries widely used in practice do have a notion of *backreferences* that allows you to do that, but that diverges from the type of regexes we're exploring in CS103. It's part of the gap between theory and practice.)

Now that we have that regex, how do we test whether it works correctly? Fortunately, we have that giant list of "yes" and "no" examples we made earlier! Ideally, we would check these automatically using a software tool rather than manually trying to see whether the

Strings and Regular Expressions provide you with a regex autograder that can do these checks for you. Out in The Real World you'd need to write your own test cases.

Split Larger Strings Into Blocks

We've wrapped up our regex for the previous language, so let's move on to a new one. Consider the following language, which we'll also call L even though it's not the same language as before:

$$L = \{ w \in \texttt{a, b, c}^\star \mid w \text{ does not contain } \texttt{abc} \text{ as a substring } \}.$$

This language is interesting in that it defines the strings in the language by giving a property they *don't* have. We want to match every string *except* for those containing `abc`. Regexes unfortunately don't have a nice way of saying "everything except for the following." This means that if we want to write a regex for this language, we'll have to figure out what shapes strings in this language *do* have. In other words, we have to convert our description from "strings that don't do X" to "strings that do do Y."

A great way to start that off is to write out a bunch of sample strings. Here's a list of strings both in and out of L:

Strings in L	Strings Not in L
`aaaaa`	`abc`
`bbbbb`	`aaabccc`
`ababbababababa`	`bcabc`
`ccccc`	`abcabc`
`aaaccc`	`cccabcccc`
`acbc`	`acbbacacbbcaacb`
ε	`cababcbac`
`abacbbbcaa`	`bbbabccc`
`acbbacbcacabcba`	`abcaaaaa`
`bc`	`abcccba`
`bcacb`	`abcabcabc`

Now that we've written things out, we need to start looking for patterns. There are many observations we can make at this point:

- Any string that doesn't use all three of `a`, `b`, and `c` is in L.
- It's always fine for a `b` to have an `a` or a `b` after it.
- It's always fine for an `a` to have an `a` or a `c` after it.
- The letter `c` can be followed by anything.
- It's only okay for an `a` to have a `b` after it if that `b` doesn't have a `c` after it.

I'm sure you can come up with more insights along these lines - the above list is just some things I've noticed after looking at the strings given above.

The question now is what to do with this. Unlike the previous example, where there were several disjoint possibilites for the string and we could just union them together, here we have a lot of different constraints that all need to be true at the same time. And that by itself is not necessarily going to be very helpful for us - regexes don't support intersections.

However, the fact that we're looking at what characters are and are not allowed to follow one another does give us a starting point. Focus on any `a` in a string in L. The rules above tell us what sorts of things we are allowed to see after it. Specifically, we can either see another `a`, or a `c`, or `b` that isn't followed by a `c`. This means that, if we want to check whether a string is in L, one option available to us would be to just focus on the positions of the `a`'s in the string and then see what happens near them. Things that happen "far away" from the most recent `a` can't change whether a string is or is not in L.

This insight - which might not be immediately obvious, but is something that comes up a lot in regex design - allows us to use a different perspective here. Given any string w that's in L, imagine splitting w apart into a bunch of smaller blocks of characters based on the positions of the letter `a`. Schematically, that might look like this:

$$\{\text{something}\}\ \texttt{a}\ \{\text{something}\}\ \texttt{a}\ \ldots\ \texttt{a}\ \{\text{something}\}\ \texttt{a}\ \{\text{something}\}$$

There's one more regex pattern we should explore in which we're alternating between {something} and `a`. This is a helpful perspective because if we can figure out how to write a regex for (something), we'll be in business. So this suggests that we might be able to write a regex for this language using the following two-step process:

1. Figure out how to express a big-picture pattern of alternating {something}'s and `a`'s.
2. Figure out how to express (something).

Let's begin with step (1). We are looking for a way to express a pattern that begins with {something}, then `a`, then {something}, then `a`, repeated as many times as we want. Whenever we see "repeat the following pattern," we should probably jump to "this is going to need a Kleene star or a Kleene plus," as those are the only ways to encode repetition in a regex.

This idea of repeating a pattern of alternating options back and forth is one that comes up a lot, and so it's worth knowing how to write this out. One option we can use is this regex:

$$\{\texttt{something}\}(\texttt{a}\{\texttt{something}\})^\star$$

How does this work? This says "lay down at least one {something}, then as many copies of "`a` {something"} as you'd like. This will indeed give us all strings we can make that have the above form.

You might notice that this forces our string to both start with {something} and to end with {something}. We should be cognizant of that, because that might not be what we want. But in this case, it's actually fine to do that. We are trying to split our string apart based on where the `a`'s are, with {something} denoting a block of characters with no `a`'s in it. If the overall string has no `a`'s, then the whole string is a {something}. If the overall string has at least one `a` in it, then we'll begin with a {something} corresponding to what occurs before the first `a`, and we'll end with a {something} corresponding to what occurs after the last `a`.

Now, what can we say about the {something}'s in the above diagram? Again, recall that we've started with some string w and split w based on the positions of the `a`'s in the string. This means that we can immediately say that whatever (something) is, it has to be made purely of `b`'s and `c`'s. Hey, that's progress!

What else can we say about the {something}s? If you look at the above observations, we can see that there are lots of rules of what is allowed to follow what. We could conceivably play around with this a bit and see if we can write a regex based purely on those requirements, but, as above, that might not be easy. We still have to combine all the requirements together, and that's not something we can do with a union.

Instead, let's apply a different one of our earlier techniques - writing out examples! Imagine you've just seen an `a` in the string, and you're wondering what combination of `b`'s and `c`'s can come afterwards. Let's write out a bunch of strings of `b`s and `c` that are permissible and a bunch of strings that aren't permissible and see if we spot anything.

Can follow `a`	Can't follow `a`
ε	`bc`
`b`	`bcb`
`c`	`bcc`
`bb`	`bcbb`
`cb`	`bcbc`
`cc`	`bccc`
`ccbcbcbbcbcbcb`	`bcbcbcbcbcbcb`
`bbcbcbcbccccb`	`bccbcbcbbcbcc`

Now, we need to look for patterns. Are there any? If we look at these strings, we can notice that there are some rules about what is allowable:

- Any string that starts with `c` is okay.
- The string `b` by itself is okay.
- Any string starting with `bb` is okay.

- The empty string is okay.

Does that account for all the possibilities? With a little review, we can see that, indeed, that really accounts for all the possibilities. And notice here that these possibilities are distinct options, which means that if we could write a regex for each one of these, we could union them all together and be done. So let's do that:

- Any string that starts with `c` is okay: $\texttt{c}(\texttt{b} \cup \texttt{c})^\star$ does the trick. This says "a `c`, followed by any string you can make out of `b`'s and `c`'s."
- The string `b` by itself is okay: `b` is a regex matching `b`.
- Any string starting with `bb` is okay: $\texttt{bb}(\texttt{b} \cup \texttt{c})^\star$ accounts for this.
- The empty string is okay: ε is a regex for the empty string.

By combining these cases together with unions, indicating that any one of these options is fine, we end up with the following regex for {something}:

$$\texttt{c}(\texttt{b} \cup \texttt{c})^\star \cup \texttt{b} \cup \texttt{bb}(\texttt{b} \cup \texttt{c})^\star \cup \varepsilon$$

That's a mouthful, but it works.

Before we move on - is there a way to simplify this? Notice that we have both $\texttt{c}(\texttt{b} \cup \texttt{c})^\star$ and $\texttt{bb}(\texttt{b} \cup \texttt{c})^\star$ in here, because any string starting with `c` is permissible and any string starting with `bb` is permissible. But there's an easier way to say this: we can just say "any string starting with either `bb` or `c` is permissible." That looks like this:

$$(\texttt{bb} \cup \texttt{c})(\texttt{b} \cup \texttt{c})^\star \cup \texttt{b} \cup \varepsilon$$

So we now have a nice way of expressing {something} - hooray! If we jump back earlier, we saw that our overall regex had the following form:

$$\{\text{something}\}(\texttt{a}\{\text{something}\})^\star$$

If we plug our regex for {something} into this expression, we get the following regex:

$$\big((\texttt{bb} \cup \texttt{c})(\texttt{b} \cup \texttt{c})^\star \cup \texttt{b} \cup \varepsilon\big)\big(\texttt{a}\big((\texttt{bb} \cup \texttt{c})(\texttt{b} \cup \texttt{c})^\star \cup \texttt{b} \cup \varepsilon\big)\big)^\star$$

That is quite a mouthful - are we sure this works? We have a nice line of reasoning that got us here, so on the one hand, we might think this is good enough and call it a day. But of course, we need to test this to make sure that we're correct. And hey, it's a good thing we have our test cases from above when we wrote out concrete strings in L and not in L, because we can run them all through our regex to see what happens.

If we run all of those strings through, we will find that they all do the right thing... with two exceptions. The string `bc` is in L, but it does ***not*** match this regex, and the same is true for `bcacb`. Uh oh! That means our answer is ***wrong*** and ***the above regex is incorrect!*** Let's take a minute to see why, because the specific error we made here is so common that it's worth calling out.

Why doesn't our regex match `bcacb`? Well, if you remember, our high-level idea behind the regex was the following:

1. Split the string at the letter `a` into blocks, giving a string of the form {something} `a` {something} `a` ... `a` {something}.
2. Figure out the requirements for what {something} can be.

Let's do this here. The string `bcacb` would be split apart into `bc`, `a`, and `cb`. We then need to see whether `bc` and `cb` match the regex we wrote for {something}. We can see that `cb` does, because it starts with `c`. However, `bc` does not match the regex. And that's intentional - we want {something} to represent "something that can follow an `a`," and `bc` can't follow an `a`.

But wait a minute - look at the string `bcacb`. Here, the `bc` ***doesn't*** follow an `a`, because it's at the start of the string, so it should be perfectly legal to have `bc` in this context. Huh.

We can similarly see why the string `bc` doesn't match this regex. If we split `bc` into blocks based on the positions of the `a`'s in the string, we'll see that we just get back the block `bc`. And, as above, the regex for {something} doesn't match this because `bc` can't legally

followed by a, but the regular expression we have doesn't follow an `a` because it's at the start of the string.

This indicates that we made a small logic error early one. The idea to split the string apart based on where `a` appears is a really good one. However, saying that this splits the string into {something} `a` {something} `a` {something} `a` ... `a` {something} is not technically correct. That would imply that the same rules apply for the characters before the first `a` and for the characters following `a`'s, which, as we've seen is not the case. A better way to split the string apart would be to write it like this:

$$\{\text{before first } \texttt{a}\}\ \texttt{a}\ \{\text{something}\}\ \texttt{a}\ \ldots\ \texttt{a}\ \{\text{something}\}$$

Recognizing that the rules for the characters before the first `a` are separate from the rules for characters after an `a` is key to cracking this regex. We can still keep all the hard work we did to figure out a regex for {something}. We now have two new tasks in front of us:

1. Figure out a regex that produces the pattern shown above: start with {before first `a`}, then `a`, then {something}, then `a`, then {something}, etc.
2. Figure out a regex for {before first `a`}.

Let's take these in turn.

For the first item, we're in luck. This sequence of blocks has a very nice pattern: it's {before first `a`}, followed by as many copies of "`a` {something}" as we'd like. We can write that partial regex like this:

$$\{\text{before first } \texttt{a}\}(\texttt{a}\ \{\text{something}\})^\star$$

Great! Now let's look at the second item: writing a regex for the strings that can appear in a string before the first `a`. But that turns out to not be too hard either. We know that any string appearing before the first `a` must consist purely of `b`'s and `c`'s (if there we an `a` there, the string would contain the first `a`, not be before it). And beyond that, there really aren't any constraints. Whatever we put here can't possibly lead to `abc` appearing in the string, because we haven't see any `a`'s yet. So we just need a regex for "all strings made of `b`'s and `c`'s," and we can do that with $(\texttt{b} \cup \texttt{c})^\star$.

All that we need to do now is put our pieces together! Taking the above regex skeleton as a starting point and plugging in our regexes for {before first `a`} and {something}, we get this regex:

$$(\texttt{b} \cup \texttt{c})^\star \left(\texttt{a}\ \left((\texttt{bb} \cup \texttt{c})(\texttt{b} \cup \texttt{c})^\star \cup \texttt{b} \cup \varepsilon\right)\right)^\star$$

Are we done? As we saw earlier, we need to run some tests on this regex. And if we run the test strings we came up with through this regex, we'll find that...

(.. drumroll ...)

... it works! Hooray!

Or at least, it works on all the test strings we came up with. Are those test strings sufficient, or did we miss something? Is there some weird edge case lurking that we didn't anticipate? That possibility always exists, but we have good reason to suspect the answer is "no." We have a good understanding of the structure of strings in L, and we've learned from our mistakes.

In CS103, you're lucky because we have a theoretically perfect autograder (e.g. an autograder that can truly check if your regexes are right by using some clever algorithms). So if this were a problem set question, you could just run the autograder and see what happens. But in practice, you'd want to make sure to really, really thoroughly test your regex before declaring victory. *Lots* of bugs in industrial code come up because someone coded up a regex wrong.

Exercises

If you're looking for some more practice using the techniques we've talked about thus far, feel free to work through the following exercises. These are purely optional and not

Collected together, these can be really rewarding.

i. Write a regex for
$L = \{w \in \{\texttt{a}, \texttt{b}\}^\star \mid w \text{ consists of either an even number of } \texttt{a}\text{'s or an odd number of } \texttt{b}\text{'s}\}$.
For example, $\texttt{aa} \in L$ and $\texttt{bbb} \in L$, but $\texttt{a} \notin L$, $\texttt{bbbb} \notin L$, and $\texttt{aba} \notin L$.

Solution

The main insight here is that every string in L is either of the form $\texttt{a}^{2n}$ or $\texttt{b}^{2n+1}$ for some choice of n. Those two options are completely separate from one another, so we can write a regex for each one independently and then union them together. Here's what that looks like:

$$(\texttt{aa})^\star \cup (\texttt{bb})^\star\texttt{b}$$

---

ii. Write a regex for $L = \{w \in \{\texttt{a}, \texttt{b}\}^\star \mid w \neq \texttt{ab}\}$.

Solution

This one is a little trickier than it looks because there is no easy way to say "everything but $\texttt{ab}$" in regex-speak. We need to figure out what structure the other strings all share that $\texttt{ab}$ doesn't.

There are a lot of ways to solve this one, but my personal favorite is to use the following observation. Any string that isn't $\texttt{ab}$ is either

- length 0, 1, or 3 or more, or
- length 2 and not $\texttt{ab}$.

We can directly encode option (1) and enumerate all choices for option (2) to get this regex:

$$\underbrace{\Sigma? \cup \Sigma^2\Sigma^+}_{\text{length 0, 1, or 3 or more}} \cup \underbrace{\texttt{aa} \cup \texttt{b}\Sigma}_{\text{length 2 and not } \texttt{ab}}$$

---

iii. Write a regex for
$L = \{w \in \{\texttt{a}, \texttt{b}, \texttt{c}\}^\star \mid |w| \geq 1 \text{ and the last character of } w \text{ appears nowhere else in } w\}$.

Solution

The main idea here is that the last character can either be $\texttt{a}$, $\texttt{b}$, or $\texttt{c}$, and based on what that character is, the preceding characters must be drawn only from the character that isn't at the end. That gives rise to the following regex:

$$(\texttt{a} \cup \texttt{b})^\star\texttt{c} \cup (\texttt{a} \cup \texttt{c})^\star\texttt{b} \cup (\texttt{b} \cup \texttt{c})^\star\texttt{a}$$

---

iv. Write a regex for $L = \{w \in \{\texttt{a}, \texttt{b}\}^\star \mid w \text{ does not contain } \texttt{aa} \text{ or } \texttt{bb} \text{ as substrings }\}$.

Solution

The major insight you need to have to solve this problem is to figure out what shape strings in L *do* have. If you write out some sample strings, you'll see that these are strings that alternate back and forth between $\texttt{a}$ and $\texttt{b}$.

There are many cases to consider for how this could happen. You could have the empty string, with no characters at all. You could have a single-character string. You could have a string starting with $\texttt{a}$ and alternating from there, or a string starting with $\texttt{b}$ and alternating from there. The

Regardless of their length, or they could have odd length.

One way to solve this would be to "brute-force" it by writing regexes for all these options and then unioning them together. That might look like this:

$$\varepsilon \cup (\texttt{ab})^\star \cup (\texttt{ba})^\star \cup (\texttt{ab})^\star\texttt{a} \cup (\texttt{ba})^\star\texttt{b}$$

However, this can be slimmed down pretty significantly. Take strings that begin with `a`, for example. They'll have some number of copies of `ab`, followed by an optional `b`. We can write "some number of copies of `ab`" as $(\texttt{ab})^\star$ and "an optional `b`" as $\texttt{b?}$. This lets us consolidate these cases further, like this:

$$\varepsilon \cup (\texttt{ab})^\star\texttt{a?} \cup (\texttt{ba})^\star\texttt{b?}$$

That's a lot shorter, but we can make this even smaller if we want. Notice that we still have ε as one of the options at the top-level. Is that actually needed? Turns out, no! You can make the empty string by taking 0 copies of `ab` and then not tacking on an optional `a`, or by taking 0 copies of `ba` and then not tacking on an optional `b`. This means that the other two options here already account for the empty string, so we can eliminate it:

$$(\texttt{ab})^\star\texttt{a?} \cup (\texttt{ba})^\star\texttt{b?}$$

But even this can be slimmed down. (Yes, really!) Here's the insight. The first part of this regex says "as many `ab`'s as you'd like, then optionally an `a`." However, not all strings start with `b`, so we have to have that second option to account for strings that start with `b`. That second option handles those strings by breaking apart a string of the form `bababababa` as `(ba)(ba)(ba)(ba)(ba)`. But we could just as easily have grouped things differently as `b(ab)(ab)(ab)(ab)a`. If you'll notice, everything in this string, except the initial `b`, is a match for the first of the two regexes in our current solution. So we can adjust our current solution by keeping the alternations of `ab`, but saying that, optionally, a `b` can come first. That gives us this:

$$\texttt{b?}(\texttt{ab})^\star\texttt{a?}$$

What a neat regex! It's super compact and elegant.

---

v. Write a regex for
$L = \{w \in \{\texttt{a}, \texttt{b}\} \mid w \text{ has an even number of copies of the substring } \texttt{ab}\}$.

Solution

The main observation needed here is that the way to get two of copies of `ab` is to have a nonempty run of `a`'s, then a nonempty run of `b`'s, then a nonempty run of `a`'s, and then another nonempty run of `b`'s. To get an even number of copies of `ab`, you need to repeat this pattern multiple times. Finally, accounting for what comes before that first run of `a`'s and after that last run of `b`'s, we can come up with this regex:

$$\texttt{b}^\star\left(\texttt{a}^+\texttt{b}^+\texttt{a}^+\texttt{b}^+\right)^\star\texttt{a}^\star$$

---

vi. Let $\Sigma = \{\begin{smallmatrix}\square\\\square\end{smallmatrix}, \begin{smallmatrix}\square\\\blacksquare\end{smallmatrix}, \begin{smallmatrix}\blacksquare\\\square\end{smallmatrix}, \begin{smallmatrix}\blacksquare\\\blacksquare\end{smallmatrix}\}$. You can think of each character in Σ as a vertical 2×1 domino where each square in the domino is either black or white.

Strings over Σ can then be thought as $2 \times n$ grids made of black and white squares. For example, here's a string of length eight, which corresponds to a 2×8 grid:

Consider the following language L over Σ:

$$L = \{ w \in \Sigma^\star \mid w \text{ is a 2D grid with a white path from its upper-left corner to its bottom-right corner } \}.$$

Here, we'll consider a path to be a series of steps moving up, down, left, or right. So, for example, the sample string above would be in L, as would the string $\begin{smallmatrix}\square & \blacksquare & \blacksquare \\ \square & \square & \square\end{smallmatrix}$. The string ε is not in L because it has no top-left or bottom-right corner. The string $\begin{smallmatrix}\square \\ \square\end{smallmatrix}$ would be in L, while $\begin{smallmatrix}\blacksquare \\ \square\end{smallmatrix}$ would not be. The string $\begin{smallmatrix}\square & \square & \square & \square & \square \\ \square & \blacksquare & \square & \blacksquare & \square\end{smallmatrix}$ would also be in L. $\begin{smallmatrix}\square & \blacksquare & \square \\ \square & \blacksquare & \square\end{smallmatrix}$ would not be in L because the $\begin{smallmatrix}\blacksquare \\ \blacksquare\end{smallmatrix}$ blocks the path. Also, $\begin{smallmatrix}\square & \blacksquare \\ \blacksquare & \square\end{smallmatrix}$ would not be in L because we cannot move diagonally.

Write a regular expression for L.

Solution

The main insight we need is that the $\begin{smallmatrix}\square \\ \square\end{smallmatrix}$ symbol serves as a separator between blocks that consist purely of $\begin{smallmatrix}\square \\ \blacksquare\end{smallmatrix}$ or $\begin{smallmatrix}\blacksquare \\ \square\end{smallmatrix}$. We'll therefore build our string by placing down at least one $\begin{smallmatrix}\square \\ \square\end{smallmatrix}$ and then filling the gaps between $\begin{smallmatrix}\square \\ \square\end{smallmatrix}$'s with runs of equal dominoes. Before the first $\begin{smallmatrix}\square \\ \square\end{smallmatrix}$ we can have as many copies of $\begin{smallmatrix}\square \\ \blacksquare\end{smallmatrix}$ as we'd like (the string can start this way), and after the last $\begin{smallmatrix}\square \\ \square\end{smallmatrix}$ we can have as many copies of $\begin{smallmatrix}\blacksquare \\ \square\end{smallmatrix}$ as we'd like (the string can end this way). Here's what that looks like:

$$\begin{smallmatrix}\square \\ \blacksquare\end{smallmatrix}^\star \begin{smallmatrix}\square \\ \square\end{smallmatrix} \left(\left(\begin{smallmatrix}\square \\ \blacksquare\end{smallmatrix}^\star \cup \begin{smallmatrix}\blacksquare \\ \square\end{smallmatrix}^\star \right) \begin{smallmatrix}\square \\ \square\end{smallmatrix} \right)^\star \begin{smallmatrix}\blacksquare \\ \square\end{smallmatrix}^\star$$

Website programming by Julie Zelenski with minor edits by Keith Schwarz and Sean Szumlanski. This page last updated 2024-Dec-04.