A Quick Guide to Regular Expressions

Professor Don Colton

Brigham Young University Hawaii

1 What Are We Trying To Do?

Regular Expressions are used for processing blocks of text. They have other uses, but we will focus on the text uses here. We will use the syntax of PERL. Similar syntax applies to TCL and other languages that support regular expressions.

A block of text is also called a **string**. It is composed of a sequence of characters. Typically we think of the string as being words designed for human view.

We have two main goals. The first goal is to recognize a string as belonging to a certain category, such as phone numbers or female names. As a human, if I show you a phone number, you are likely to be able to identify it as a phone number, even if you have never seen that phone number before.

The second goal is to extract useful information from the string. This information may be located and identified by its surrounding context.

There can be other goals, such as the modification of the string to cause it to meet certain standards. For instance, we could find all instances of two or more spaces in a row and replace each instance with a single space. Or we could look for common spelling errors and replace them with their corrections.

Because these types of problems occurs again and again, computer scientists have been motivated to find a good solution. The solution is called **regular expressions**.

2 Recognition

The first goal is to recognize a string as being a member of a certain category. Is it a social security number? Is it a vehicle license plate number? Is it a possible word in English? As a human, you are likely to be able to answer yes or no in each case with a high degree of accuracy. Let's begin by focusing on license plates.

2.1 License Plates

In Hawaii currently a vehicle license plate generally consists of three letters followed by three digits. There are other configurations, including so-called vanity plates that can be almost anything. Let's focus on typical plates. Say your computer program input is `ABC123` and you must determine if that meets the pattern. As a human you can easily say yes. But as a computer program, how easy is it? One could build a program with a lot of **if** statements, like this:

```
# Plan A: check every plate
# about 17.5 million lines long
$okay = 0; # default
if ( $plate eq "AAA000" ) { $okay = 1 }
if ( $plate eq "AAA001" ) { $okay = 1 }
if ( $plate eq "AAA002" ) { $okay = 1 }
... you can guess what goes in between
if ( $plate eq "ZZZ999" ) { $okay = 1 }
```

This would be a really long program. About 17.5 million lines. You would probably like to avoid writing it. So here is an alternative that looks at each character separately.

```
# Plan B: check every letter
# about 127 lines long
$char6 = chop ( $plate );
$okay6 = 0;
if ( $char6 eq "0" ) { $okay6 = 1 }
if ( $char6 eq "1" ) { $okay6 = 1 }
...
if ( $char6 eq "9" ) { $okay6 = 1 }
$char5 = chop ( $plate );
$okay5 = 0;
if ( $char5 eq "0" ) { $okay5 = 1 }
...
$char1 = chop ( $plate );
$okay1 = 0;
if ( $char1 eq "A" ) { $okay1 = 1 }
if ( $char6 eq "B" ) { $okay1 = 1 }
```

```
...
if ( $char6 eq "Z" ) { $okay1 = 1 }
$okay = 1; # default
if ( $char1 != 0 ) { $okay = 0 }
...
```

Validating one digit takes 12 lines. Validating one letter takes 28 lines. That totals 127 lines. You would probably agree it is a big improvement over Plan A above.

2.2 Character Classes

To recognize a digit, we can use the following regular expression.

```
if ( $ch6 =~ m/[0123456789]/ ) { $ok6 = 1 }
```

This deserves some explanation. The first interesting thing is the `=~` operator, which in PERL indicates that a regular expression is being matched or substituted. (It can also indicate a translation operation, but that is beyond the scope of this paper.)

The `m/../` part indicates that a matching operation is happening. The `m` means match. (An `s` is used for substitution.) The slashes are delimiters that surround the regular expression.

The `[0123456789]` part indicates a character class. It matches when any of the characters in that list matches.

This gives rise to yet another plan.

```
# Plan C
# about 14 lines long
$ok = 0;
$ch6 = chop ( $plate );
if ( $ch6 =~ m/[0123456789]/ ) { $ok++ }
$ch5 = chop ( $plate );
if ( $ch5 =~ m/[0123456789]/ ) { $ok++ }
$ch4 = chop ( $plate );
if ( $ch4 =~ m/[0123456789]/ ) { $ok++ }
... do something for the letters
if ( $ok == 6 ) { success! }
```

You can see that this is a big improvement over Plan B. It shortens the testing of each digit by nine lines. It shortens the program considerably. The result is about 14 lines, which almost 90% smaller than 127 lines.

2.3 Character Ranges

It is common to want to match a character to an uninterrupted range of characters, such as zero to nine, or A to Z. Regular expressions typically include a short-cut way to express the same thought given above by using a dash to indicate a character range.

```
if ( $ch6 =~ m/[0-9]/ ) { $ok6 = 1 }
```

Characters (digits and letter and punctuation) are stored in the computer as a sequence of binary digits (bits: zeros and ones). One typical code is called ASCII. Another is UNICODE. Character ranges work because the digits zero to nine take up adjacent positions in the character code.

```
0 in ASCII is 0110000
1 in ASCII is 0110001
2 in ASCII is 0110010
3 in ASCII is 0110011
4 in ASCII is 0110100
5 in ASCII is 0110101
6 in ASCII is 0110110
7 in ASCII is 0110111
8 in ASCII is 0111000
9 in ASCII is 0111001
```

If you are familiar with binary numbers, you will see that the ASCII codes for "0" to "9" are the binary numbers for 48 through 57. The main thing is that the range includes everything we do want and nothing we don't want.

Thinking about letters, we might have considered doing this:

```
$ch1 =~ m/[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/
```

Fortunately the letters A to Z form a range. We can say `[A-Z]`. And little a to little z form a range (`[a-z]`). Unfortunately those two ranges are separated so we cannot say `[A-z]` and have it work correctly.

Here is an improved program

```
# Plan D
# about 14 lines long
$ok = 0;
$ch6 = chop ( $plate );
if ( $ch6 =~ m/[0-9]/ ) { $ok++ }
$ch5 = chop ( $plate );
if ( $ch5 =~ m/[0-9]/ ) { $ok++ }
$ch4 = chop ( $plate );
if ( $ch4 =~ m/[0-9]/ ) { $ok++ }
$ch3 = chop ( $plate );
if ( $ch3 =~ m/[A-Z]/ ) { $ok++ }
$ch2 = chop ( $plate );
```

```
if ( $ch2 =~ m/[A-Z]/ ) { $ok++ }
$ch1 = chop ( $plate );
if ( $ch1 =~ m/[A-Z]/ ) { $ok++ }
if ( $ok == 6 ) { success! }
```

2.4 Matching Several Characters

Regular expressions are not limited to matching a single character at a time. We can match several characters at the same time. In this regular expression, we are looking for letter, letter, letter, digit, digit, digit.

```
$plate =~ m/[A-Z][A-Z][A-Z][0-9][0-9][0-9]/
```

Notice that we eliminate the chops and we can make the whole process happen in a single statement:

```
if ( $plate =~ m/.../ ) { success! }
```

This is an incredible improvement, but language tinkerers seem to never be satisfied until they have squeezed the last drop of redundancy out of something.

2.5 Multipliers

Here is another improvement.

```
if ( $plate =~ m/[A-Z]{3}[0-9]{3}/ ) ...
```

This improvement uses `{3}` as a multiplier to say that `[A-Z]` must happen exactly three times, and `[0-9]` must also happen exactly three times.

We can use the multiplier to give us some flexibility on the length. Here is a regular expression to match a license plate that consists of between 1 and 7 letters or digits in any order.

```
if ( $plate =~ m/[A-Z0-9]{1,7}/ ) ...
```

Notice that we have included two character ranges in the character class, and we have given a multiplier of `{1,7}` which means a minimum of 1 and a maximum of 7 repetitions.

To match three or more of something, we can say `{3,}`. To match less than five of something, we can say `{0,4}`.

There are three special cases that are so popular they have an even shorter short cut.

```
{0,}   is   *
{1,}   is   +
{0,1}  is   ?
```

When something can be repeated zero or more times, we put a star (*) in the regular expression. For one or more times, we can put a plus (+). If something is optional, meaning it can occur zero or one times, we can put a question mark (?).

Here is a regular expression for a number with a decimal point:

```
[+-]?[0-9]+[.][0-9]+
```

Notice it has an optional sign (`[+-]?`), followed by one or more digits (`[0-9]+`), followed by a dot (`[.]`), followed by one or more digits. If we are using dash (`-`) itself as one of the characters, we simply list it last (or first) in the character class.

2.6 A Small Lie

It is time to correct a small oversight in our discussion. To keep things simple, we have not talked about position anchors. There are two of them commonly used with regular expressions. The start-of-string anchor is caret (`^`). The end-of-string anchor is dollar (`$`).

Without these position anchors, all our regular expressions mentioned above will match any string that **contains** the target. For instance,

```
$line =~ m/[+-]?[0-9]+[.][0-9]+/
```

will match any `$line` that has a decimal point number anywhere inside. To restrict our attention to lines that have nothing else before the number, we can say this:

```
$line =~ m/^[+-]?[0-9]+[.][0-9]+/
```

Notice the caret at the front of the regular expression. To restrict our attention to lines that have nothing else **after** the number, we can say this:

```
$line =~ m/[+-]?[0-9]+[.][0-9]+$/
```

To restrict our attention to lines that have nothing else before or after the number, we can say this:

```
$line =~ m/^[+-]?[0-9]+[.][0-9]+$/
```

2.7 More Shortening

By the way, the `m` in the `m/.../` construct is optional. We can leave it out if we are using slashes as the delimiters. It is another short cut.

Also, there is a short cut for digit. Instead of saying `[0-9]`, we can simply say `\d`. There are a number of similar short cuts.

3 Data Extraction

Say we want the middle two digits of a US Social Security number. We can recognize the number like this:

```
/^\d{3}-\d{2}-\d{4}$/
```

This means: Must begin at the start of the string. Match three digits. Match one dash. Match two digits. Match another dash. Match four digits. Must end at the end of the string.

By placing parenthesis around the part we wish to extract, we can isolate parts of the string that interest us.

```
$ssn =~ /^\d{3}-(\d{2})-\d{4}$/
$middle = $1;
```

In this case, there are parenthesis around the `\d{2}`. Because these are the first set of parenthesis, the matching part is stored in special variable `$1`.

Here is a longer example:

```
$ssn = "321-54-9876";
$ssn =~ /^(\d{3})-(\d\d)-(\d{4})$/;
$first = $1;
$middle = $2;
$last = $3;
```

In this case, `$first` will end up containing `321`. `$last` will end up containing `9876`.

3.1 Nuts and Bolts

A problem arises in CGI programming. We will not discuss CGI programming in any depth here, but suffice it to say that we can receive inputs that look like this:

```
nuts=5&bolts=12
```

We can use a regular expression to extract the 5 and 12 which we may need to use to update a database or fulfil an order. We can do it like this:

```
chomp ( $line = <STDIN> );
$line =~ /^nuts=(\d+)&bolts=(\d+)$/;
$nuts = $1;
$bolts = $2;
```