Relationship Between DFA and GREP
Martha Kosa

When you program, you need to use variables to store information. You declare a variable by defining its type and name; the valid type names and punctuation vary by programming language. Whatever language you use to program, you should give your variables descriptive names so that your code is easier to debug, both for you and for others who may be working with your code. Older programming languages such as FORTRAN only allowed short variable names due to computer memory constraints back then. More modern programming langugages allow variable names of arbitrary length; the variable names can contain both letters (uppercase and lowercase) and digits (and possibly underscores and dollar signs, depending on the language). We will keep it simple and just consider variable names with lowercase letters for now.

Let's design a DFA to accept valid variable names containing only lowercase letters. Variable names must have at least one letter.

Try It!

1. If JFLAP is not already active, start JFLAP and click the **Finite Automaton** button.
2. How many states do you need for the machine? Create the states, and make the first one be a starting state.
3. Draw the necessary transitions. Notice how the alphabet [¥] changes as new letters appear.
4. Which of your states should be a final state? Make it be a final state.
5. How many transitions have you drawn? Are you tired? ☺
6. Verify that the empty string is rejected by your DFA.
7. Verify that a single letter is accepted by your DFA.
8. Verify that your first name is accepted by your DFA. How many steps are needed for acceptance?
9. Verify that your last name is accepted by your DFA. How many steps are needed for acceptance?
10. Verify that **name** is accepted by your DFA. How many steps are needed for acceptance?

Your resulting DFA should look similar to the following.

Try It!

1. Choose a sample string containing a digit. Verify that the string is rejected by your DFA/ How many steps are needed before rejection occurs?
2. How many transitions are needed to accept strings containing alphabetic or

numeric characters, where the numeric characters cannot appear at the beginnings of strings?
3. Draw those transitions. Notice how [illegible] changes as new digits appear.
4. Verify that your sample string containing a digit above is not accepted by your DFA. How many steps are needed for acceptance?

Your resulting automaton should look like the following:

It took you a long time to draw your automaton above because you had a large alphabet, first 26 symbols for the letters, and then 10 more symbols for the digits, for a grand total of 36 letters.

It turns out that every DFA can be converted into an equivalent regular expression; JFLAP has a tool to do that.

Try It!

1. Select ***Convert > Convert to RE***. If you are in a hurry, you can click the **Step to Completion** button and then the **Export** button.
2. You will need to resize the window and use the scrollbars and/or the **Table Text Size** slider to see the entire regular expression.

Your resulting expression should look like the following.

Questions to Think About

1. What regular expression operator corresponds to the + in the regular expression?
2. What regular expression operator(s) allow(s) for variable names of arbitrary length?

Although it is simple, this regular expression is very long because of the size of its alphabet. Is there a shorthand notation? Yes, there is. It is called GREP-style notation. "**G**lobal Search for **R**egular **E**xpression and **P**rint" is the translation for the acronym GREP. It comes from the Unix operating system world.

Remember the base regular expressions ([illegible], [illegible], and each a [illegible][illegible]) and the building blocks for regular expressions: concatenation, union, and Kleene star for repetition.

In GREP, we are wanting to match strings, such as file names or patterns in documents, so we don't use [illegible], a placeholder for the empty set. The empty string [illegible] will be used indirectly in an optional choice.

Symbol juxtaposition (a fancy word for placing items next to each other in the desired order) is used as before for concatenation. The * is used as before for repetition.

The symbol | is used to denote union.

How do we get more concise notation? Left and right square brackets ([and], respectively) are used to represent the union of alphabet symbols. When we use letters and numbers, we can use range expressions. They exploit the ASCII (American Standard Code for Information Interchange) numeric codes to represent characters. The ASCII codes for the lowercase letters range from 97 for the lowercase 'a' to 122 for the lowercase 'z', while the ASCII codes for the uppercase letters range from 65 for the uppercase 'A' to 90 for the uppercase 'Z'. The ASCII codes for the digts range from 48 for the digit '0' to 57 for the digit '9'.

Here are some examples:

[abc] represents an a, b, or a c.

- [a-z] represents any possible lowercase letter.
- [UDLR] represents an 'U', a 'D', an 'L', or an 'R', perhaps useful in a navigation application with arrow keys
- [a-zA-Z] represents any lowercase or uppercase letter
- [02468] represents any even digit
- [0-9] represents any digit
- [a-zA-Z0-9] represents any alphanumeric character
- [abc]* represents any number of a's, b's, and/or c's, in any combination and in any order
- [a-z]* | [A-Z]* represents any string consisting of all lowercase letters or all uppercase letters

Try It!

1. Rewrite the regular expression corresponding to our first automaton in GREP-style notation.
2. Rewrite the regular expression corresponding to our second automaton in GREP-style notation.

When we work with variables in our computer programs, we need to initialize them. If our variables have been declares as integers, we will need to initalize at least some of them with integer values. What is the proper format for an integer literal (constant)? We need at least one digit. What happens if we have multiple digits? We don't want 0's at the beginning (leading zeroes) unless we are actually representing the number 0.

Try It!

1. Design a DFA to accept the set of unsigned nonnegative integers without leading

zeros.

2. Select ***Convert > Convert to RE***. If you are in a hurry, you can click the **Step to Completion** button and then the **Export** button.
3. Rewrite the regular expression corresponding to the automaton in GREP-style notation.

Your expression converted from your automaton should look similar to the following.

An integer can be positive or negative. We use a minus sign (-) to denote a negative integer. If we don't include a minus sign, we assume that the number is nonnegative. We can easily modify the above regular expression to account for negative integers. 0 should not have a sign preceding it. The minus sign, if it exists, will precede the integer. Either the minus sign exists, **or** it doesn't. What special symbol corresponds to the string containing nothing (aka the empty string)? What regular expression operation corresponds to the **or** logical operation? The union operation does; how is it expressed in GREP-style notation?

Try It!

1. Edit your regular expression above to allow for the possibility of negative integers. Select ***Convert > RE to FA*** to see the resulting automaton. Remember that every finite automata has a corrsponding regular expression and vice versa. (NOTE: This does not work properly in JFLAP 8.0 – I tried with both (! + -) and (- + !) before the expression. It also does not work properly in JFLAP 7.0 – I got the error dialog "Lambda does not cat with anything else".)
2. Edit your automaton to accept the set of all integers without leading 0's. Only negative integers will have a minus sign preceding them. You may need to add another state, and you may need to use a lambda transition.

Your automaton should look similar to the following.

Try It!

1. Select ***Convert > Convert to RE***. You will get an error message. What kind of conversion is needed first before the automaton can be converted to an equivalent regular expression? Do that conversion first, and then let JFLAP guide you through the rest.
2. Rewrite the regular expression corresponding to the automaton in GREP-style notation.
3.

Your expression converted from your automaton should look similar to the following.

Notice that your expression has a lot of repetition. In particular, the subexpression **1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9** is appears twice at the beginning before the subexpression with the Kleene star. We can use GREP-style notation to simplify the expression to something like the following:

(-[1-9] | [1-9])[0-9]* | 0

We can do even more. What is happening? We either have 0 or 1 copies of the minus sign followed by the subexpression [1-9]. GREP -style notation gives us "x?" to denote 0 or 1 copies of expression x. We can now simplify further to produce the following:

[-]?[1-9][0-9]* | 0

Let's do one more example. When you are driving in your car, you need a license plate for your car with a unique identifier. Each state has a specified format for its license plate identifiers. A typical one is 3 capital letters followed a – and then followed by 4 digits.

Try It!

1. Design a DFA to accept the set of valid license plate identifiers as described above.
2. How many states does your DFA have?
3. How many final states does it have?
4. How many transitions does it have?
5. How many strings does it accept?
6. Rewrite the regular expression corresponding to your DFA in GREP-style notation.

Your automaton should look similar to the following. In a production automaton, we would need all possible capital letters and all possible digits in the transitions.

Your converted regular expression in JFLAP should look similar to the following.

When you convert this to GREP-style notation, you will have several repeated subexpressions. The [A-Z] and the [0-9] subexpressions are repeated, four times for the [A-Z] and three times for the [0-9].

GREP-style notation gives us the following shortcuts for expressing repeated patterns:

- x{n} allows for x to occur n times
- x{n,} allows for x to occur at least n times

- x{m,n} allows for x to occur at least m times and at most n times
- x+ allows for x to occur one or more times

Now you can simplify the expression for valid license plates to be elegant and concise. It will be [A-Z]{4}-[0-9]{3}. Depending on your regular expression system, you may need to use an escape symbol before the middle – symbol.

Try It!

1. How many valid license plates are possible based on the GREP-style regular expression just shown.
2. Suppose your state's population grows and you need to allow for 3 or 4 capital letters before the – symbol and 3 or 4 digits afterward. Write the GREP-style regular expression describing all possible valid license plates.
3. How many valid license plates are possible based on the GREP-style regular expression you just wrote?

Congratulations! You have explored the relationship between finite automata and GREP-style regular expressions, and have seen the power of the GREP-style notation. You can visit http://www.regex101.com to explore regular expressions further, as used to validate strings in PHP, JavaScript, and Python. You can use GREP-style notation. Many employers ask questions about regular expressions in job interviews, so you have some important motivation for practing with them.