Regular expressions

A regular expression (r.e.) is a way of describing a language. It can consist of a finite set of words, like

\{cat, dog, mouse, ε\}

which represents the 4 words that are listed. Then, there are 3 operators that can appear in r.e’s: they are

- Union
- Concatenation
- Closure

these operators allow you to glue together subexpressions to form larger expression.

Union and Concatenation

**Union:** Any r.e. represents a set of words (its language) and we may use ∪ or \{·, ·, ·\} to connect 2 or more subexpressions into a larger regular expression, e.g.

\{cat, dog, mouse, ε\} ∪ \{cat, cats\}

which represents the 5 words ε, cat, cats, dog, mouse.

**Concatenation**

By joining two r.e’s together, we denote the set of words you can make by taking a word from the first r.e. and concatenating it to some word from the second r.e., e.g.:

\{over, under\}\{cooked, state, rate\}

denotes the words overcooked, undercooked, overstate, understate, overrate, underrate.

Closure (also called “Kleene closure”)

If we take a regular expression and add the superscript *, we get a new r.e. that represents the set of all words you can make by taking any sequence of words from the original r.e. and concatenating them together.

Example:

\{(cat)\}*

denotes the words ε, cat, catcat, catcatcat, catcatcatcat, ...

Notice that using closure, you can define an infinite language!

Examples

\{(cat, dog)\}*

denotes the words ε, cat, dog, catcat, catdog, dogcat, dogdog, catcatcat, catcatdog, catdogcat, ...

\a\{(a, b)\}*

is the set of all words over alphabet \{a, b\} that begin with the letter a.

We may write this as \a\{(a, b)\}*, using the convention that closure is applied to the shortest regular subexpression that precedes it.
Examples

\{a, b\}^* \{c, d\}^*

denotes words such as \texttt{abababccdddd}, \texttt{abbbabbbaaaccddddd}, ...

Write down a r.e. for strings of \texttt{a}'s and \texttt{b}'s where all the \texttt{a}'s must come before all the \texttt{b}'s.

Write down an expression for the above language, assuming that the empty word is not allowed to belong to the language.

Common extensions to the notation

Let \( E \) denote a regular expression. \((E)^n\) denotes concatenations of \(n\) words generated by \( E \). (Could be written \( EEE ... E \) \((n \text{ times})\) 

\((E)^+\) denotes concatenations of at least one word generated by \( E \) (Could be written \( E E^* \) or \( E^* E \)).

The above give no extra expressive power in terms of what languages can be described.

A variant of the "union" notation:

\( E_1 \cup E_2 \) can be written as \( \{E_1, E_2\} \) (the latter is similar to javascript reg exp syntax); generally \( E_1 \cup E_2 \cup ... \cup E_r \) can be written as \( \{E_1, E_2, ..., E_r\} \)

Lexical tokens

We can use regular expressions for short, precise definitions of lexical tokens:

```
‘variable: string of letters/digits starting with a letter’
```

r.e.: \( \{a,b, ..., z,A,B, ..., Z\} \{a,b, ..., z,A,B, ..., Z, 0,1,2, ..., 9\}^* \)

number in exponential/scientific notation ("1.16121122E-03" denotes \(1.16121122 \times 10^{-3}\), i.e. \(0.00116121122\))

\( \{1, 2, 3, ..., 9\}, \{0, 1, 2, ..., 9\}\)\^\(8\)\{00, \{\epsilon, -\}\{\{1, 2, ..., 9\}\{0, 1, 2, ..., 9\}, \{0, 1, 2, ..., 9\}\{1, 2, ..., 9\}\}}

\(\{a,b\}^* \{a,a\}^*\) represents strings containing \texttt{aaa} as substring.

Assuming that alphabet \( A = \{a, b\} \), the above can be written as \( A^* \texttt{aaa} A^* \).

\(b^* ab^* ab^* ab^*\): strings with 3 \texttt{a}'s.

\(A^* A^* a A^* a A^*\): strings with at least 3 \texttt{a}'s.

How about strings with an even number of \texttt{a}'s and any number of \texttt{b}'s? (there’s a 2-state FA that accepts this language.)
Using regular expressions for lexical analysis

Use regular expressions to specify lexical tokens. A parser generator can convert r.e’s to DFAs. Given a sequence of characters that may or may not be a token, it is easy to check whether it is accepted by the DFA. (Notice that it is not straightforward to check whether a string of symbols belongs to the language given by a regular expression.

A compiler, given a program, will look for a prefix that corresponds to a token, when that prefix is found, add it to list of tokens, delete it from program, and repeat, on remainder of program. (A lexical error is generated if no prefix matches any token expression.)

A language that is easier to describe using a regular expression

Alphabet $A = \{0, 1, 2, \ldots, 9\}$

$L \subseteq A^*$ is defined to be words which contain an even number of occurrences of at least one of the digits $0, 1, 2, \ldots, 9$

So the word $0123456789$ (or any permutation of it) is the shortest word not in $L$.

The regular expression
\[
\{ \{1, 2, \ldots, 9\}^*0\{1, 2, \ldots, 9\}^* \}^* \cup \{1, 2, \ldots, 9\}^*
\]
defines all words with an even number of $0$’s.

Take union of that with 9 similar expressions (for the other digits) and you have a r.e. for $L$. A DFA would need $2^{10}$ states to keep track of whether each digit has appeared an odd or an even number of times so far.

A language that is easier to describe using a DFA

Alphabet $A = \{0, 1, 2, \ldots, 9\}$

$L \subseteq A^*$ is defined to be numbers whose digits add up to a multiple of 10. e.g. $1234, 28, 2828$.

Regular Languages...

...are those languages that can be described using regular expressions. We will see that these are the same languages as those that can be accepted by finite automata (Kleene’s theorem).

In terms of description length some languages are “better” expressed as DFAs and some as reg exprs. r.e’s usually give a easy-to-understand description, on the other hand it’s obvious how to run DFAs on input strings.