Regular Languages

The following are regular languages over the alphabet $\Sigma$:
- $\emptyset$ (empty language)
- $\{\}$ (empty string)
- $\sigma$ (any element in the alphabet $\Sigma$)
- $\{\sigma\}$ (set containing the element $\sigma$)
- $A \cup B$ (union of any regular languages $A$ and $B$)
- $A \cup B$ (concatenation of any regular languages $A$ and $B$)
- $\alpha^+ = \{\epsilon\} \cup A \cup AA \cup AAA \cup \ldots$ (repetition of regular language $A$)

Notice that all finite languages are regular, but not all infinite languages.

Regular languages do not allow arbitrary “nesting” (e.g. parens).

Formal Languages

In formal language theory:
- Alphabet: any set (usually a character set, like English or ASCII)
- Letter: an element in the given alphabet, e.g. "a"
- String (or word): finite sequence of letters, e.g. "hi"
- Language: a set of strings, e.g. \{"a", "aa", "aaa", \ldots\}

We might omit the quotes/braces, so we'll use the following denotations:
- $\epsilon$: empty string (i.e., "")
- $\delta$: empty language (i.e., empty set \{\})
A regular expression is an easier way to describe a regular language. It's essentially a pattern for describing a regular language.

For example, in \([ab-c-z]\)\(^*(1+2|3)\)?\(^4\)?\(^7\), we have:
- \([ab-c-z]\) (a character set) means "either a, b, c, u, x, y, or z".
- Asterisk (a.k.a. "Kleene star"); a quantifier means "zero or more"
- Plus (another quantifier) means "one or more"
- Question mark (another quantifier) means "at most one"
- Backslash ("escape") before a special character means it's a literal character
- Pipe (the OR symbol) means "either", and parentheses group

Notice the transition function \(\delta\) outputs a set with exactly one state (a singleton).

In a deterministic finite automaton (DFA), the transition function always outputs a set with exactly one state (a singleton).

In a nondeterministic finite automaton (NFA), the above is not true.

### Regular Expressions

Python has a regex engine to find text matching a regex:

```python
>>> import re
>>> n = n = re.match(r'[^0-9-]+\d[^0-9-]+', 'hello cs61a@berkeley.edu cs698-52')
>>> n
<re.Match object; span=(0, 24),
match='hello cs61a@berkeley.edu'>
```

Notice that these could all be handled by re.match:
- Substring search (`str.find`)
- Subsequence search (`re.match`)

The grep tool (from ed’s g/regex/p — global/regex/print) does this for files.

### Million-dollar question:

How do you find text matching a regex?

Two steps:
- Parse the regex (pattern) to "understand" its structure
- Use the regex to parse the actual text (corpus)

It turns out that:
- Step 1 is theoretically harder, but practically easier.
  (This can be done similarly to how you parsed Scheme.)
- Step 2 is theoretically easier, but practically harder.

This is because we need parsing the corpus to be fast.

### Finite Automata

A finite automation (FA) consists of the following (example below)\(^2\):

- An input alphabet \(\Sigma = \{0, 1\}\) (here)
- A finite set of states \(S = \{s_0, s_1, s_2\}\) (here)
- An initial state \(s_0 \in S\) (here)
- A set of accepting (or final) states \(F \subseteq S = \{s_2\}\) (here)
- A transition function \(\delta : S \times \Sigma \rightarrow 2^S\) (the arrows here)

Notice the transition function \(\delta\) outputs a subset of states.

In a deterministic finite automaton (DFA), the transition function always outputs a set with exactly one state (a singleton).

i.e., in a DFA, the next state is determined by the input & current state. (i.e., every state has exactly 1 arrow leaving it for each possible input.)

In a nondeterministic finite automaton (NFA), the above is not true.

### Finite Automata

Finite automata are language recognizers: you feed a string as an input, and if it accepts the input string, the string is in its language.\(^3\)

In particular:
- \(\Rightarrow\) Finite automata recognize regular languages, and nothing else

Therefore, we can:
- Convert regex pattern to FA
- Feed corpus to FA in linear time!
- ...
- Profit!

But how can we do this?

---

\(^1\)From wikipedia: A long-enough input must contain a repeatable substring. (Why?)

\(^2\)Note that as FA is not quite the same thing as a finite-state machine (FSM).

\(^3\)Million-dollar question: How do you find text matching a regex?
Consider: \((a|b)^*(1+2|3)\). Ask: Where in the pattern can we be?

- \(s_0 = \bullet(a|b)^*(1+2|3) = \bullet(a|b)^*(1+2|3) = \bullet(a|b)^*\cdot(1+2|3)\)
- \(s_1 = (a|b)\cdot(1+2|3) = (a|b)\cdot(1+2|3)\cdot = (a|b)^*\cdot(1+2|3)\)
- \(s_2 = (a|b)^*(1+2|3)\cdot = (a|b)^*(1+2|3)\cdot = (a|b)^*\cdot(1+2|3)\cdot = (a|b)^*\cdot(1+2|3)\cdot\cdot\cdot\)

(Expanding a state to its equivalents is a mathematical closure operation.)

### Conclusion

This is just the tip of the iceberg for string algorithms (and automata). Languages, grammars, and automata are also used in computational linguistics, computational biology/genomics (DNA alignment/matching)...

It is extremely easy to graduate and avoid languages & automata. But they provide the keys for solving many otherwise difficult problems. You can see more in EE/CS 144, 149, 151, 164, 172, 219C, 291E...

--- Related Words of Wisdom ---

- Kleene is next to Godelness.

Thank you!