MAU22C00 Lecture 28

John Stalker

Trinity College Dublin

Intersection (reminder)

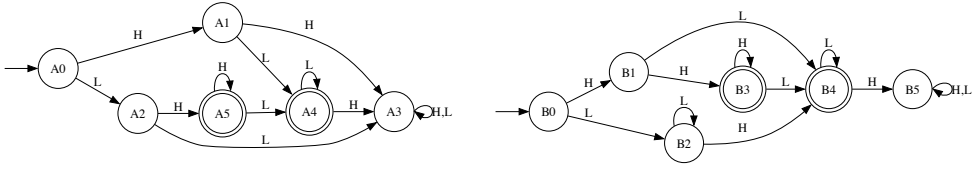

Figure 1: FSAs for individual languages

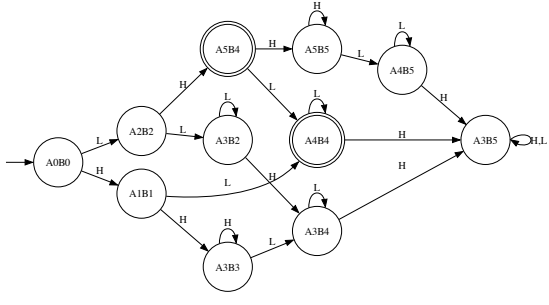

Figure 2: FSA for the intersection

Relative complements

We can also construct finite state automata for the relative complement of two languages. The following FSA recognises those strings which belong to Tokyo language but not the Kansai one.

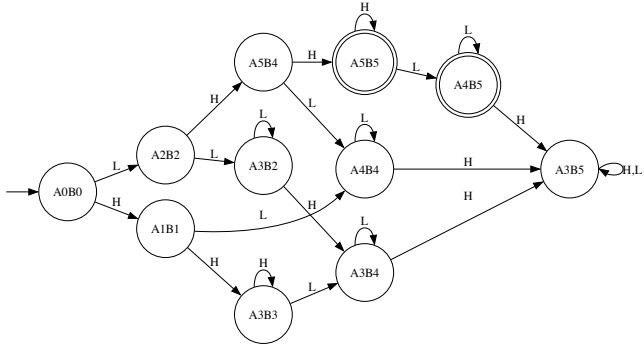

Figure 3: An FSA for Tokyo but not Kansai

The accepting states are the combinations of an accepting state for Tokyo with a rejecting one for Kansai.

Relative complements, continued

We can, of course, also do this in the reverse dirction.

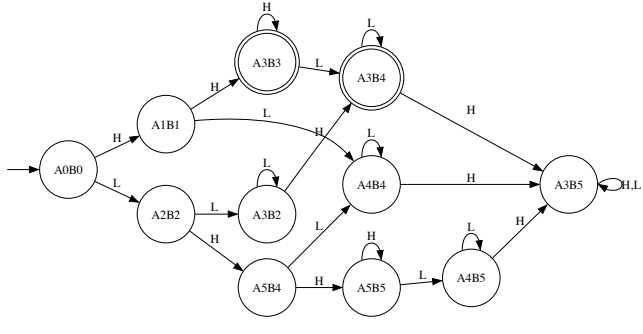

Figure 4: An FSA for Kansai but not Tokyo

Relative complements (subtleties)

The procedure I used to get a recogniser for the relative complement was this:

- Start from finite state automata recognising the two languages
- Create a new finite state automaton
  - whose states are (reachable) pairs of states for the two original automata
  - whose initial state is the pair of initial states
  - whose transitions for each pair on a given token are computed by looking at each FSA separately
  - whose accepting states are those which are a pair with an accepting state and a rejecting state

Question: Does this work in general?

Answer: No! There are two problems:

- If the FSAs are non-deterministic one of them might get to a rejecting state for some input even though it could also have got to an accepting state.
- If the FSAs are deterministic but not strongly deterministic one of them might never get to the end of the input.

Relative complements (more subtleties)

This construction can only be guaranteed to work if the automata you start from are strongly deterministic.

Fortunately we've already given an algorithm, the power set algorithm, which takes a finite state automaton and gives you a strongly deterministic one which recognises the same language.

There were two versions of the power set algorithm, one where we used only the reachable sets–good for practical applications–and one where we just threw in all of them–good for theoretical purposes.

There are also two versions of the product construction, the one we did in the last lecture, one where we use only the reachable pairs–good for practical applications–and one where we just throw in all of them–good for theoretical purposes.

Using the practical versions gave me an FSA with 11 states. Using the theoretical versions would have given me one with 262,144 states.

Application?

Japanese is a pitch accent language with two tones, high and low.

Different regional accents have different pitch accent patterns.

If you know them you can figure out where someone is from by noticing patterns which occur in one accent but not another.

The main regional accents are the Tokyo and Kansai accents. The regular languages in the recurring example are (a slightly simplified version of) the pitch patterns for words in those accents.

Regular expressions

Regular expressions are yet another way of describing a simple class of languages.

The following are regular expressions:

- `(HLL*)|(LHH*L*)`: This will match any string which is either an `HL` followed by any number of `L`'s or an `LH` followed by any number of `H`'s and then any number of `L`'s.
- `(HHH*L*)|(HLL*)|(LL*HL*)`: This will match any string which is an `HH` followed by any number of `H`'s and then any number of `L`'s or an `HL` followed by any number of `L`'s, or at least one `L` followed by an `H` and then any number of `L`'s.

The first of these is an alternate description of the Tokyo language and the second is an alternate description of the Kansai language.

Regular expression syntax

- Characters represent themselves, unless they are special, in which case they need to be escaped with a preceding backslash \.
- Concatenation is represented by concatenation, e.g. `HL` matches an `H` followed by an `L`.
- An asterisk, `*`, is used for Kleene star, i.e. some (possibly zero) repetitions of whatever preceded it. So `L*` matches any string of `L`'s and `LL*` matches any non-empty string of `L`'s.
- A vertical line represents alternates, as with phrase structure grammars. So `H|L` matches an `H` or an `L`, while `|H` matches an empty string or an `H`, i.e. an optional `H`.
- Parentheses are used for grouping. So `(H|L)*` matches a string of `H`'s and `L`'s while `H|(L*)` matches either a single `H` or any string of `L`'s.
- The special characters include \, `*`, `|`, (, and).

In theory we need a separate notation for a regular expression matching nothing.

An example

Can we build a regular expression which matches our (normalised) integers?

- An integer is either zero or non-zero.
- A non-zero integer is an optional - followed by a positive integer.
- A positive integer is a non-zero digit followed by any number of digits.

We can build this up incrementally.

- `0` matches zero.
- `|-` matches an optional -
- `(1|2|3|4|5|6|7|8|9)` matches a non-zero digit.
- `(0|1|2|3|4|5|6|7|8|9)` matches a digit.
- `(0|1|2|3|4|5|6|7|8|9)*` matches any number of digits.

An example, continued

- `0` matches zero.
- `|-` matches an optional -
- `(1|2|3|4|5|6|7|8|9)` matches a non-zero digit.
- `(0|1|2|3|4|5|6|7|8|9)` matches a digit.
- `(0|1|2|3|4|5|6|7|8|9)*` matches any number of digits.
- `(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*` matches a positive integer.
- `(|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*` matches a non-zero integer.
- `0|((|-)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*)` matches any integer.

Versions of regular expressions

Different people mean different things by the term "regular expression".

- In theoretical discussions people mean what I've just defined. The only operations are Kleene star, concatenation and union (alternation). This is the original meaning of the term. The notation may vary from author to author though.
- Regular expressions are ubiquitous in practical programming, anywhere that pattern matching is needed. Usually extra special characters are added, like `?` for optional items, or `+` for Kleene plus, i.e. one or more repetitions, which could be expressed using the three basic operations. If you do a lot of pattern matching you should learn these extensions.
- Some implementations, e.g. PCRE, add special characters for operations which cannot be expressed in terms of those three operations. You cannot rely on anything we say in this module being true in those implementations.

Closure properties

The union of two languages described by regular expressions can be described by a regular expression. Just put a | between them, and add parentheses if needed.

The concatenation of two languages described by regular expressions can be described by a regular expression. Just concatenate the regular expressions, and add parentheses if needed.

The Kleene star of a language described by a regular expression can be described by a regular expression. Just add a $*$ at the end, and add parentheses if needed.

The definition of regular expressions was designed to make these operations easy, but other operations are possible.

The reversal of a language described by a regular expression can be described by a regular expression. Just reverse the order of all concatenations.

Intersections and relative complements are trickier, and require a different point of view.