Regular Expressions cont'd; Applications to string and pattern matching

Informatics 2A: Lecture 6

Mary Cryan

School of Informatics
University of Edinburgh
`mcryan@inf.ed.ac.uk`

28 September 2018

Recap of Lecture 5

- Finished off the details of the marking algorithm for minimizing DFAs, plus discussed the alternative approach of Brzozowskis's "double reversal" (without details).
- Regular languages are closed under the operations of concatenation and Kleene star.
- This is proved using ϵ-NFAs, which can be easily converted to ordinary NFAs.
- (still to do) Regular expressions provide a textual representation of regular languages.
- (still to do) Kleene's Theorem: regular expressions define exactly the regular languages.
- We won't prove the difficult half of Kleene's theorem formally, but equation solving using Kleene algebra and Arden's Rule gives the idea.

Regular expressions

We've been looking at ways of specifying regular languages via machines (often presented as pictures). But it's very useful for applications to have more textual ways of defining languages.

A regular expression is a written mathematical expression that defines a language over a given alphabet Σ.

- The basic regular expressions are

$$\emptyset \qquad \epsilon \qquad a \ \ (\text{for } a \in \Sigma)$$

- From these, more complicated regular expressions can be built up by (repeatedly) applying the two binary operations $+$, $.$ and the unary operation * . Example: $(a.b + \epsilon)^* + a$

We use brackets to indicate precedence. In the absence of brackets, * binds more tightly than ., which itself binds more tightly than $+$.

$$\text{So} \quad a + b.a^* \quad \text{means} \quad a + (b.(a^*))$$

Also the dot is often omitted: ab means $a.b$

How do regular expressions define languages?

A regular expression is itself just a written expression. However, every regular expression α over Σ can be seen as defining an actual language $\mathcal{L}(\alpha) \subseteq \Sigma^*$ in the following way.

- $\mathcal{L}(\boldsymbol{\emptyset}) = \emptyset, \quad \mathcal{L}(\boldsymbol{\epsilon}) = \{\epsilon\}, \quad \mathcal{L}(a) = \{a\}.$
- $\mathcal{L}(\alpha + \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$
- $\mathcal{L}(\alpha.\beta) = \mathcal{L}(\alpha)\,.\,\mathcal{L}(\beta)$
- $\mathcal{L}(\alpha^*) = \mathcal{L}(\alpha)^*$

Example: $a + ba^*$ defines the language $\{a, b, ba, baa, baaa, \ldots\}$.

The languages defined by $\boldsymbol{\emptyset}, \boldsymbol{\epsilon}, a$ are obviously regular.

What's more, we've seen that regular languages are closed under union, concatenation and Kleene star.

This means every regular expression defines a regular language. (Formal proof by induction on the size of the regular expression.)

Exercises

Consider (again) the language

$$\{x \in \{0,1\}^* \mid x \text{ contains an even number of 0's}\}$$

Which of the following regular expressions define the above language?

1. $(1^*01^*01^*)^*$
2. $(1^*01^*0)^*1^*$
3. $1^*(01^*0)^*1^*$
4. $(1 + 01^*0)^*$

Exercises

Consider (again) the language

$$\{x \in \{0,1\}^* \mid x \text{ contains an even number of 0's}\}$$

Which of the following regular expressions define the above language?

1. $(1^*01^*01^*)^*$
2. $(1^*01^*0)^*1^*$
3. $1^*(01^*0)^*1^*$
4. $(1 + 01^*0)^*$

Answer: 2 and 4 define the required language. 1 doesn't: e.g. 11 doesn't match the expression. 3 doesn't: e.g. 00100 doesn't match the expression.

Kleene's theorem

We've seen that every regular expression defines a regular language.

Remarkably, the converse is also true: every regular language can be defined by a regular expression.

The equivalence between regular languages and expressions is:

Kleene's theorem

> *DFAs and regular expressions give rise to exactly the same class of languages (the regular languages).*

(For proof, see Kozen, Lecture 9.)
As we've already seen, NFAs (with or without ϵ-transitions) also give rise to this class of languages.

So the evidence is mounting that the class of regular languages is mathematically a very natural and well-behaved one.

Kleene algebra

Regular expressions give a textual way of specifying regular languages. This is useful e.g. for communicating regular languages to a computer.

Another benefit: regular expressions can be manipulated using algebraic laws (Kleene algebra). For example:

$$
\begin{array}{rclcrcl}
\alpha + (\beta + \gamma) & = & (\alpha + \beta) + \gamma & \quad & \alpha + \beta & = & \beta + \alpha \\
\alpha + \emptyset & = & \alpha & & \alpha + \alpha & = & \alpha \\
\alpha(\beta\gamma) & = & (\alpha\beta)\gamma & & \boldsymbol{\epsilon}\alpha & = & \alpha\boldsymbol{\epsilon} \ = \ \alpha \\
\alpha(\beta + \gamma) & = & \alpha\beta + \alpha\gamma & & (\alpha + \beta)\gamma & = & \alpha\gamma + \beta\gamma \\
\emptyset\alpha & = & \alpha\emptyset \ = \ \emptyset & & \boldsymbol{\epsilon} + \alpha\alpha^* & = & \boldsymbol{\epsilon} + \alpha^*\alpha = \alpha^*
\end{array}
$$

Often these can be used to simplify regular expressions down to more pleasant ones.

Other reasoning principles

Let's write $\alpha \leq \beta$ to mean $\mathcal{L}(\alpha) \subseteq \mathcal{L}(\beta)$ (or equivalently $\alpha + \beta = \beta$). Then

$$\begin{aligned} \alpha\gamma + \beta \leq \gamma &\quad\Rightarrow\quad \alpha^*\beta \leq \gamma \\ \beta + \gamma\alpha \leq \gamma &\quad\Rightarrow\quad \beta\alpha^* \leq \gamma \end{aligned}$$

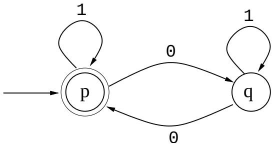
Arden's rule: Given an equation of the form $X = \alpha X + \beta$, its smallest solution is $X = \alpha^*\beta$.

What's more, if $\epsilon \notin \mathcal{L}(\alpha)$, this is the *only* solution.

Beautiful fact: The rules on this slide and the last form a complete set of reasoning principles, in the sense that if $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$, then '$\alpha = \beta$' is provable using these rules. (Beyond scope of Inf2A.)

DFAs to regular expressions

We use an example to show how to convert a DFA to an equivalent regular expression.



For each state r, let the variable X_r stand for the set of strings that take us from r to an accepting state. Then we can write some simultaneous equations:

$$\begin{aligned} X_p &= 1X_p + 0X_q + \epsilon \\ X_q &= 1X_q + 0X_p \end{aligned}$$

Where do the equations come from?

Consider:

$$X_p = 1X_p + 0X_q + \epsilon$$

This asserts the following.

Any string that takes us from p to an accepting state is:

- a 1 followed by a string that takes us from p to an accepting state; or
- a 0 followed by a string that takes us from q to an accepting state; or
- the empty string.

Note that the empty string is included because p is an accepting state.

Solving the equations

We solve the equations by eliminating one variable at a time:

$$\begin{array}{rrcll} & X_q & = & 1^*0X_p & \text{by Arden's rule} \\ \text{So} & X_p & = & 1X_p + 01^*0X_p + \epsilon & \\ & & = & (1 + 01^*0)X_p + \epsilon & \\ \text{So} & X_p & = & (1 + 01^*0)^* & \text{by Arden's rule} \end{array}$$

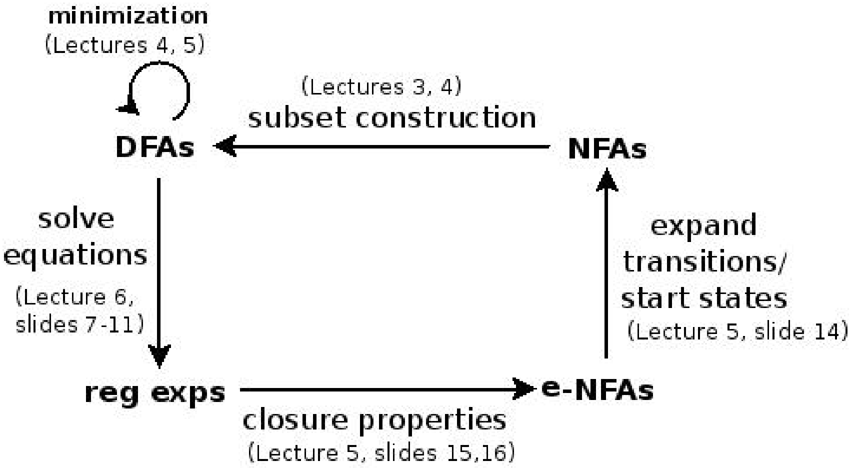
Since the start state is p, the resulting regular expression for X_p is the one we are seeking. Thus the language recognised by the automaton is:

$$(1 + 01^*0)^*$$

The method we have illustrated here, in fact, works for arbitrary NFAs (without ϵ-transitions).

Theory of regular languages: overview

How it all fits together . . .



Applications of regular language technology

We'll now start to look at several practical applications of the theory we've covered:

- efficient string searching
- more general pattern searching
- data validation, e.g. for XML documents
- lexical analysis for computer languages (first stage of the language processing pipeline)
- automated verification of safety and liveness properties for complex interacting systems — typically via model checking.

Further applications will be discussed in the Natural Language part of the course.

String and pattern matching with Grep tools

Important practical problem: Search a large file (or batch of files) for specific strings, or strings of a certain form.

Most UNIX/Linux-style systems since the '70s have provided a bunch of utilities for this purpose, known as Grep (Global Regular Expression Print).

Extremely useful and powerful in the hands of a practised user. Make serious use of the theory of regular languages.

Typical uses:

```
grep "[0-9]*\.[0-9][0-9]" document.txt
```

-- searches for prices in pounds and pence

```
egrep "(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)" document.txt
```

-- searches for occurrences of the word "the"

grep, egrep, fgrep

There are three related search commands, of increasing generality and correspondingly decreasing speed:

- `fgrep` searches for one or more fixed strings, using an efficient *string matching* algorithm.
- `grep` searches for strings matching a certain pattern (a simple kind of regular expression).
- `egrep` searches for strings matching an extended pattern (these give the full power of regular expressions).

All three of these make use of the ideas we've been studying.

Efficient string matching

Suppose we want to search for occurrences of a shortish string s in a very long document D.

Obvious method: For each position p within D, check whether there's an occurrence of s starting at p, by working through s one character at a time until:

- either there's a character mismatch
- or we reach the end of s (search successful).

(Example on next slide.)

Can we do better?

Naive string search: an example

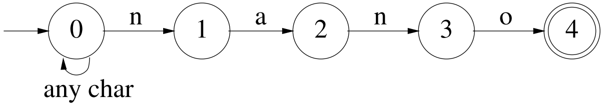
Suppose we're searching for nano within a long document D containing nanobananas.

n	a	n	o	b	a	n	a	n	a	s
n	a	n	✓							
	X									
		n	X							
			X							
				X						
					X					
						n	a	n	X	
							X			
								n	a	X
									X	
										X

Notice that several characters in D are visited more than once.

Better method: The Knuth-Morris-Pratt algorithm

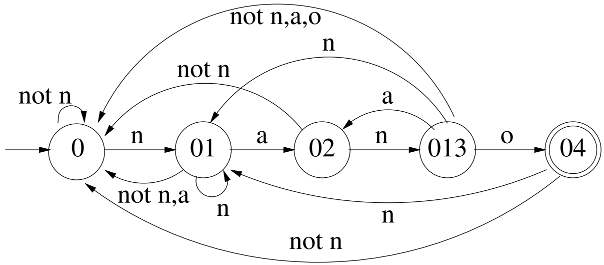
The following NFA clearly accepts all strings ending in nano:



So we can . . .

1. First convert this to an equivalent DFA *M*. (Costs some time—but worth it if *s* is short and *D* is very long.)
2. Run *D* through *M*. (Each character of *D* processed just once; no buffering required.)
3. Every time we enter an accepting state of *M*, signal a hit.

The corresponding DFA



- For NFAs of this kind, the subset construction behaves nicely: no state explosion. Can even optimize the construction a bit for this class of NFAs.
- Useful in practice for documents with a lot of repetition. E.g. imagine searching for `www.inf.ed.ac.uk/inf2a/` in a long list of web addresses, where most begin with `www.` and many begin with `www.inf.ed.ac.uk/`

Optional exercise

1. Suppose now that the search string is nana. Construct the appropriate DFA in this case.
2. Convince yourself that this will detect all occurrences of nana, even overlapping ones!

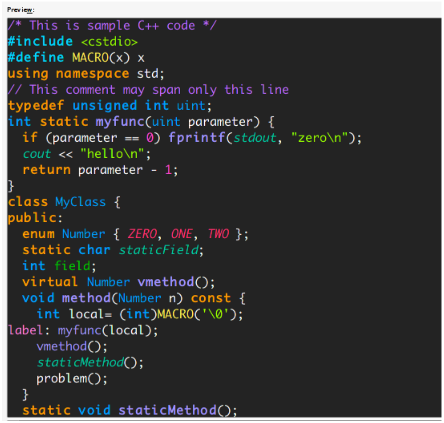
Reading

Relevant reading:

- Regular expressions: Kozen chapters 7,8; J & M chapter 2.1.
- From regular expressions to NFAs: Kozen chapter 8; J & M chapter 2.3.
- Kleene algebra: Kozen chapter 9.
- From NFAs to regular expressions: Kozen chapter 9.

Next time: pattern matching, lexical analysis

We will first consider how we search for “one of a group” of fixed strings. Then we will continue to pattern matching via `egrep` and `grep` (using r.ex. symbols).



```cpp
/* This is sample C++ code */
#include <cstdio>
#define MACRO(x) x
using namespace std;
// This comment may span only this line
typedef unsigned int uint;
int static myfunc(uint parameter) {
  if (parameter == 0) fprintf(stdout, "zero\n");
  cout << "hello\n";
  return parameter - 1;
}
class MyClass {
public:
  enum Number { ZERO, ONE, TWO };
  static char staticField;
  int field;
  virtual Number vmethod();
  void method(Number n) const {
    int local= (int)MACRO('\0');
label: myfunc(local);
    vmethod();
    staticMethod();
    problem();
  }
  static void staticMethod();
```

How does the colouring work?