CITS2211 Discrete Structures
Regular Expressions

Unit coordinator: Débora Corrêa
Lecturer: Matt Ryan

Highlights

What we know so far:

- FSMs accept certain input strings
- The set of all input strings accepted by an FSM is called the **language** recognised by that FSM
- So we can specify languages using techniques we already know for **sets**
- But that is not very convenient.

Today we will study:

- a **compact way to describe** the language recognised by a FSM by using their equivalence with **regular expressions**.

Why study regular expressions

- Regular expressions (REs) are widely used in Computer Science
- REs are used to search text for strings that match certain patterns
- Linux grep, awk, Perl, text editors all provide regular expression mechanisms
- Programming languages such as Java, Python provide regular expression libraries

Reading

Introduction to the Theory of Computation by M Sipser

Chapter 1: Regular Languages
Section 1.3 Regular Expressions

Regular Expressions

Lecture Outline: Regular Expressions

In this class we will study

- Regular expressions
- Regular languages
- We will see that not all languages are regular

Regular Expressions

We need a compact way to describe the languages that can be recognised by finite state machines.

Define a **regular expression** over an alphabet Σ as follows:

1. The symbol $\varnothing$ and the symbol ϵ are regular expressions
2. Any symbol $i \in \Sigma$ is a regular expression
3. If A and B are regular expressions, then so are (AB), $(A + B)$ and $(A)^*$

Notice that this is a recursive definition, very similar to the definition of a well-formed formula. Brackets are only for grouping and can be omitted if the resulting expression is unambiguous.

Regular Languages

Associated with every regular expression is a collection of strings known as a **regular set** or **regular language**.

1. The regular expression $\varnothing$ represents the empty language $\varnothing$
2. The regular expression ϵ represents the set $\{\epsilon\}$ containing only the empty string
3. The regular expression i represents $\{i\}$
4. The regular expression AB represents the set of all strings of the form $\alpha\beta$ where $\alpha \in A$ and $\beta \in B$.
5. The regular expression $A + B$ represents the union of the regular sets represented by A and B
6. The regular expression $(A)^*$ represents the set of all concatenations of strings in the regular set represented by A.

Examples of regular languages

1. 0^* represents the set $\{\epsilon, 0, 00, 000, 0000, \ldots\}$
2. $0 + 1^*$ represents the set $\{0, \epsilon, 1, 11, 111, 1111, \ldots\}$
3. 0^*1^* represents the set

$$\left\{ \begin{array}{lllll} \epsilon & 1 & 11 & 111 & 1111 & \ldots \\ 0 & 01 & 011 & 0111 & 01111 & \ldots \\ 00 & 001 & 0011 & 00111 & 001111 & \ldots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{array} \right\}$$

4. $(0^*)(0 + 1^*)$ represents the set

$$\left\{ \begin{array}{llllll} 0 & \epsilon & 1 & 11 & 111 & 1111 & \ldots \\ 00 & 0 & 01 & 011 & 0111 & 01111 & \ldots \\ 000 & 00 & 001 & 0011 & 00111 & 001111 & \ldots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{array} \right\}$$

Non-regular languages

Q: Are all languages regular ?
A: No

We can use a *diagonalization argument* from earlier in the course to show that there are non-regular languages. In fact, there are even some non-regular languages over the alphabet $\{0, 1\}$.

Firstly, notice that the set of regular expressions is *countable*— we can arrange them in lexicographical order. Therefore we can form a complete list R_1, R_2, R_3, ..., of regular expressions.

Let L_1, L_2, ..., be the corresponding list of regular languages described by the above regular expressions.

Now we will form a new language L and show that it is not in the list above.

The diagonalization argument

Recall that the earlier diagonalization arguments formed a new object that was definitely *different* from every object on the supposedly complete list.

We will copy this strategy as follows:

Consider the set of strings 1, 11, 111, 1111, and form the new language L as follows:

If $1 \notin L_1$, then add it to L (else do nothing)
If $11 \notin L_2$, then add it to L
If $111 \notin L_3$, then add it to L
If $1111 \notin L_4$, then add it to L and so on...

Then clearly L is different from L_1, L_2, ... and thus L is not regular.
Either L_i contains $111\ldots11$ (i times) and L does not, or vice versa.

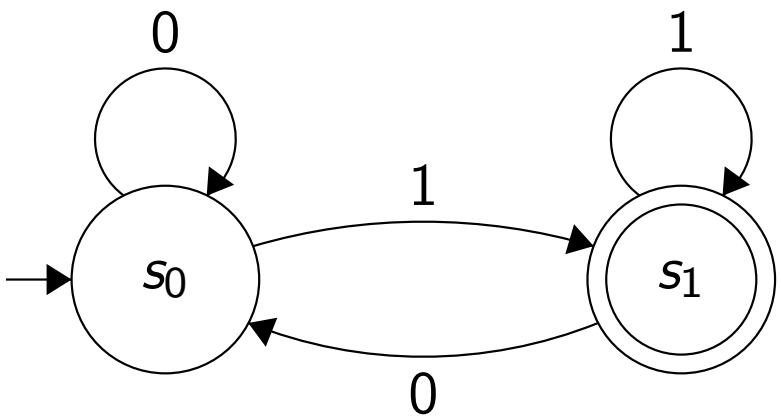
This is an example of a *non-constructive* proof. We have shown the *existence* of a non-regular language, but have not actually *specified* one.

FSMs and regular languages

Let us consider several FSMs, and see if we can work out what regular language they recognize, and whether we can identify a regular expression for it.

FSMs and regular languages

Consider the following recogniser with start state s_0 and accepting state s_1:



What language does this machine recognise?

FSMs and regular languages

We can start by looking at some input strings and seeing if there is a pattern...

Accepted:

`1`, `01`, `001`

Rejected:

`0`, `00`, `10`

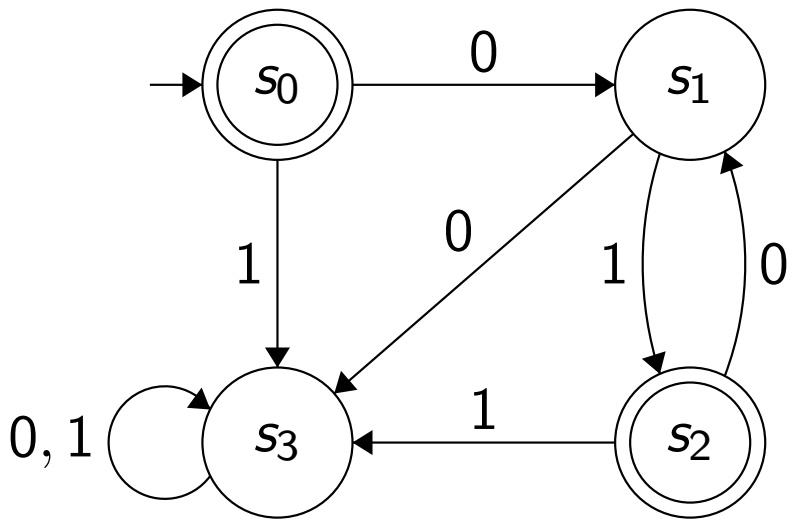
We hypothesise that this FSM accepts strings ending in 1, and a moment's thought shows that this is indeed correct.

Therefore the language accepted by this FSM is

$$L(M) = (0 + 1)^*1$$

Another example

Write down some accepted strings for this FSM. And some rejected ones. Can you see a pattern?



Another example, cont'd

For this example we can see that if the FSM ever reaches state s_3 then it can never reach the accepting states (s_0 or s_2). Therefore any accepted string must start with 0.

After than, another 0 will change the state to s_3, and therefore the second character must be a 1.

We continue this argument to see that the FSM will only be in an accepting state provided it has received a sequence of 01 pairs. As soon as this pattern is broken, the machine changes to state s_3.

Therefore the language accepted by this FSM is

$$L(M) = (01)^*$$

Kleene's theorem

Kleene's theorem

The languages recognised by the FSMs given earlier could all be described with regular expressions

Kleene's theorem shows that this is no coincidence

Stephen Kleene, 1909-94, American

Amongst many achievements, Kleene invented regular expressions

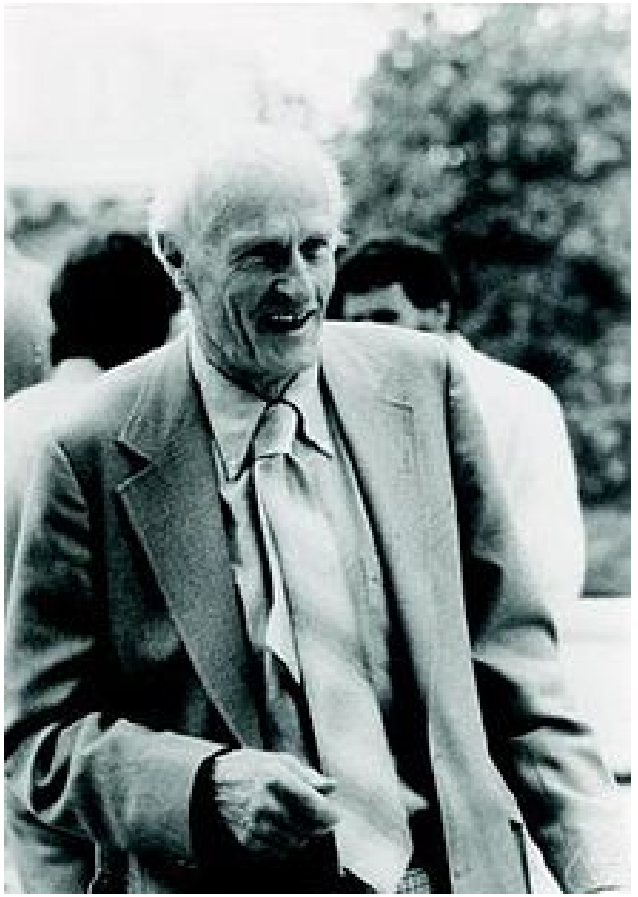


Photo source: wikipedia

An application of Kleene's theorem

Theorem: **Kleene's Theorem**
A language is regular if and only if it is the language recognised by some finite state machine.

Before we consider the proof of Kleene's theorem, we consider an application of it:

We will prove that the language

$$L = \{01, 0011, 000111, \ldots\} = \{0^n 1^n : n \in \mathbb{N} \wedge n > 0\}$$

is *not* regular.

Kleene's theorem, cont'd

We will use a proof by contradiction, whereby we assume that the language *is* regular, and show that this leads to a contradiction.

Proof: If L is regular, then there is some finite state machine M that recognises L. Suppose that M has m states. Now consider the operation of M on the input strings

$$0, 00, 000, \ldots, 0^{m+1}$$

As M has only m states, the pigeonhole principle implies that (at least) two of these strings (of length v and w), say 0^v, 0^w leave M in the same state. But then because M accepts the string 0^v1^v it must also accept the string 0^w1^v which is not in L since the numbers of 0s and 1s are different since $w \neq v$

Therefore there is no finite state machine that recognises L. ■

Proof of Kleene's theorem

It is necessary to show two things in order to prove Kleene's theorem:

1. Given a regular expression, we can find a finite state machine to accept the strings described by that regular expression
2. Given a finite state machine, we can find a regular expression to describe the strings accepted by that machine

Regular expression $\rightarrow$ FSM

Given a regular expression, we wish to find a finite state machine that can recognise the corresponding language.

This is hard to do directly, and so is done in two steps:

- Finding a non-deterministic finite state machine that recognises the appropriate language
- Converting the non-deterministic finite state machine into a deterministic one.

The concept of *non-deterministic* computation is surprisingly powerful in theoretical computer science (even though not practicable in reality, barring quantum computing).

Finite state machine $\rightarrow$ regular expression

This is also a two-stage process:

- Convert the finite state machine into an equivalent generalised non-deterministic finite state machine
- Reduce the number of states of this machine down to a single state labelled with the corresponding regular expression.

This procedure is tedious, but mechanical and so we omit the details. See the Sipser text.

Summary

What we know now

- Regular expressions are a compact notation for specifying languages (sets of strings)
- Every regular expression determines a regular language
- Not all languages are regular
- A language is regular iff it is the language recognised by some FSM (Kleene's theorem)