University of Portsmouth
BSc (Hons) Computer Science
Third Year

Theoretical Computer Science (THEOC)
M21276
September 2025 - January 2026
20 Credits

Thomas Boxall
`thomas.boxall1@myport.ac.uk`

Contents

Page 1

Lecture - A1: Introduction to Languages

2025-09-29 14:00 Janka

There are two useful decks of slides on Moodle: Introduction to THEOCs and Overview of THEOCs. There is also a *Worksheet 0* which recaps the key concepts from year 1's Architecture & Operating Systems (Maths) and 2nd year's Discrete Maths and Functional Programming.

1.1 Introduction

Languages are a system of communication. The languages we commonly use are built for communicating and passing along instructions to other humans or computers. Depending on the context in which a language is used, will vary the precision which must exist within the language. For example, a language to convey "pub tonight?" to a friend can be as simple as that, where the human can add context clues to fill in the blanks; however to convey `print('hello, world')` to a computer - the language must be precise as it is not designed to interpret sloppy writing.

Languages are defined in terms of the set of symbols (called it's *alphabet*), which get combined into acceptable *strings*, which happens based on rules of sensible combination called *grammar*.

We can take this definition and see it in practice for the English Language:

Alphabet The alphabet for the English Language is Latin: $A = \{a, b, c, d, e, \ldots, x, y, z\}$

Strings (words) Strings are formed from A, for example 'fun', 'mathematics'. The English vocabulary defines which are really strings (for example which appear in the Oxford English Dictionary)

Grammar From the collection of words, we can build sentences using the English rules of *grammar*

Language The set of possible sentences that make up the English Language

Whilst this is an example around a tangible, understandable example - the elements of a formal language are exactly the same however they must be defined without any ambiguity. For example, programming languages have to be defined with a precise description of the syntax used.

1.2 Formalising Language Definitions

Definitions

Alphabet A finite, nonempty set of symbols. For example: $\Sigma = \{a, b, c\}$

String A finite sequence of symbols from the alphabet (placed next to each other in juxtaposition). For example: abc, aaa, bb are examples of strings on Σ

Empty String A string which has no symbols (therefore zero length), denoted Λ.

Language Where Σ is an alphabet, then a language over Σ is a set of strings (including empty string Λ) whose symbols come from Σ

For example, if $\Sigma = \{a, b\}$, then $L = \{ab, aaab, abbb, a\}$ is an example of a language over Σ.

Languages are not finite and they may or may not contain an empty string.

If Σ is an alphabet, then Σ^* denotes the infinite set of all strings made up from Σ - including an empty string. For example, if $\Sigma = \{a, b\}$ then $\Sigma^* = \{\Lambda, a, b, ab, aab, aaab, bba, \ldots\}$. We can therefore say that when looking at Σ^*, a language over Σ is any subset of Σ^*.

Example: Languages

For a given alphabet, Σ, it is possible to have multiple languages. For example:

- $\emptyset$ - an empty language
- $\{\Lambda\}$ - a language containing only an empty string (silly language)
- Σ - the alphabet itself
- Σ^* - the infinite set of all strings made up from the alphabet

Alternatively, we can make this slightly more tangible:
Where $\Sigma = \{a\}$:

- $\emptyset$
- $\{\Lambda\}$
- $\{a\}$
- $\{\Lambda, a, aa, aaa, aaaa, \ldots\}$

1.3 Combining Languages

It is possible to combine languages together to create a new language.

1.3.1 Union and Intersection

As languages are just sets of strings, we can use the standard set operations for Union and Intersection to combine the languages together.

Example: Union and Intersection

Where $L = \{aa, bb, ab\}$ and $M = \{ab, aabb\}$
Intersection (common elements between the two sets): $L \cap M = \{ab\}$
Union (all elements from each set): $L \cup M = \{aa, bb, ab, aabb\}$

1.3.2 Product

The product of two languages is based around concatenation of strings...

The operation of *concatenation of strings* places two strings in juxtaposition. For example, the concatenation of the two strings *aab* and *ba* is the string *aabba*. We use the name *cat* to denote this operation: cat(aab, ba) = aabba. We can combine two languages L and M by forming the set of all concatenations of strings in L with strings in M, which is called the product of two languages.

Definitions

Product of two languages If L and M are languages, then the new language called the product of L and M is defined as $L \cdot M$ (or just LM). This can be seen in set notation below:

$$L \cdot M = \{cat(s, t) : s \in L \text{ and } t \in M\}$$

The product of a language, L, with the language containing only an empty string returns L:

$$L \cdot \{\Lambda\} = \{\Lambda\} \cdot L = L$$

The product of a language, L, with an empty set returns an empty set:

$$L \cdot \emptyset = \emptyset \cdot L = \emptyset$$

The operation of concatenation is not commutative - meaning the order of the two languages matters. For two languages, it's usually true that:

$$L \cdot M \neq M \cdot L$$

Example: Commutativity Laws of Concatenation

For example, if we take two languages: $L = \{ab, ac\}$ and $M = \{a, bc, abc\}$

$$\begin{aligned} L \cdot M &= \{aba, abbc, ababc, aca, acbc, acabc\} \\ M \cdot L &= \{aab, aac, bcab, bcac, abcab, abcac\} \end{aligned}$$

They have no strings in common!

The operation of concatenation is associative. Which means that if L, M, and N are languages:

$$L \cdot (M \cdot N) = (L \cdot M) \cdot N$$

Example: Associativity Laws of Concatenation

For example, if $L = \{a, b\}$, $M = \{a, aa\}$ and $N = \{c, cd\}$ then:

$$\begin{aligned} L \cdot (M \cdot N) &= L \cdot \{ac, acd, aac, aacd\} \\ &= \{aac, aacd, aaac, aaacd, bac, bacd, baac, baacd\} \\ \text{which is the same as} & \\ (L \cdot M) \cdot N &= \{aa, aaa, ba, baa\} \cdot N \\ &= \{aac, aacd, aaac, aaacd, bac, bacd, baac, baacd\} \end{aligned}$$

1.3.3 Powers of a Language

For a language, L, the product $L \cdot L$ is denoted by L^2.

The language product L^n for ever $n \in \{0, 1, 2, \ldots\}$ is defined as follows:

$$\begin{aligned} L^0 &= \{\Lambda\} \\ L^n &= L \cdot L^{n-1}, \text{if } n > 0 \end{aligned}$$

Example: Powers of Languages

If we take $L = \{a, bb\}$:

$$\begin{aligned} L^0 &= \{\Lambda\} \\ L^1 &= L = \{a, bb\} \\ L^2 &= L \cdot L = \{aa, abb, bba, bbbb\} \\ L^3 &= L \cdot L^2 = \{aaa, aabb, abba, abbbb, bbaa, bbabb, bbbba, bbbbbb\} \end{aligned}$$

1.4 Closure of a Language

Attempt to extricate a better definition of closure out of Janka

The *closure* of a language is an operation which is applied to a language.

If L is a language over Σ (for example $L \subset \Sigma^*$) then the closure of L is the language denoted by L^* and is defined as follows:

$$L^* = L^0 \cup L^1 \cup L^2 \cup \dots$$

The *Positive Closure* of L is the language denoted by L^+ and is defined as follows:

$$L^+ = L^1 \cup L^2 \cup L^3 \cup \dots$$

So from this we can derive that $L^* = L^+ \cup \{\Lambda\}$. However - it's not necessarily true that $L^+ = L^* - \{\Lambda\}$.

For example, if we take our alphabet as $\Sigma = \{a\}$ and our language to be $L = \{\Lambda, a\}$ then $L^+ = L^*$.

Based on what we now know, there's some interesting properties of closure we can derive. Let L and M be languages over the alphabet Σ:

- $\{\Lambda\}^* = \emptyset^* = \{\Lambda\}$
- $L^* = L^* \cdot L^* = (L^*)^*$
- $\Lambda \in L$ if and only if $L^+ = L^*$
- $(L^* \cdot M^*)^* = (L^* \cup M^*)^* = (L \cup M)^*$
- $L \cdot (M \cdot L)^* = (L \cdot M)^* \cdot L$

These will be explored more in the coming Tutorials

1.5 Closure of an Alphabet

As we saw earlier, Σ^* is the infinite set of all strings made up from Σ. The closure of Σ coincides with our definition of Σ^* as the set of all strings over Σ. In other words, it is a nice representation of Σ^* as follows:

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

From this, we can see that Σ^k represents the set of strings of length k, for each their symbols are in Σ.

Page 2

Lecture - A2: Grammars

2025-09-29 15:00 Janka

As we saw in the previous lecture, languages can be defined through giving a set of strings or combining from the existing languages using operations such as productions, unions, etc. Alternatively, we can use a grammar to define a language.

To describe a grammar for a language - two collections of alphabets (symbols) are necessary.

Definitions

Terminal Symbols from which all strings in the language are made. They are symbols of a 'given' alphabet for generated language. Usually represented using lower case letters

Non-Terminal Temporary Symbols (different to terminals) used to define the grammar replacement rules within the production rules. They must be replaced by terminals before the production can successfully make a valid string of the language. Usually represented using upper case letters.

Now we know what terminals & non-terminals are - we need to know how to produce terminals from non-terminals. This is where the *Production Rules* come into play. Productions take the form:

$$\alpha \to \beta$$

where α and β are strings of symbols taken from the set of terminals and non-terminals.

A grammar rule can be read in any of several ways:

- "replace α by β"
- "α produces β"
- "α rewrites to β"
- "α reduces to β"

We can now define the grammar.

Definitions

Grammar A set of rules used to define a language - the structure of the strings in the language. There are four key components of a grammar:

1. An alphabet T of symbols called *terminals* which are identical to the alphabet of the resulting language
2. An alphabet N of grammar symbols called *non-terminals* which are used in the production rules
3. A specific non-terminal called the *start symbol* which is usually S
4. A finite set of *productions* of the form $\alpha \to \beta$ where α and β are strings over the alphabet $N \cup T$

2.1 Using a Grammar to Generate a Language

Every grammar has a special non-terminal symbol called a *start symbol* and there must be at least one production with left-side consisting of only the start symbol. Starting from the production rules with the start symbol, we can step-by-step generate all strings belonging to the language described by a given grammar.

As we begin converting from Non-Terminal to Terminal containing strings, we introduce the *Sentential Form*. As we continue to generate strings, we introduce *derivation*.

Definitions

Sentential Form A string made up of terminal and non-terminal symbols.

Derivation Where x and y are sentential forms and $\alpha \to \beta$ is a production, then the replacement of α with β in $x\alpha y$ is called a derivation. We denote this by writing:

$$x\alpha y \Rightarrow x\beta y$$

During our derivations, there are three symbols we may come across:

- $\Rightarrow$ derives in one step
- $\Rightarrow^+$ derives in one or more steps
- $\Rightarrow^*$ derives in zero or more steps

Finally, we can define $L(G)$: the Language defined by the given Grammar.

Definitions

$L(G)$ If G is a grammar with start symbol S and set of terminals T, then the language generated by G is the following set:

$$L(G) = \{s | s \in T^* and S \Rightarrow^+ s\}$$

Great - now we've seen the theory, lets put it into an example.

Example: Using a Grammar to Derive a Language

Let a grammar, G, be defined by:
- the set of terminals $T = \{a, b\}$
- the only non-terminal start symbol S
- the set of production rules: $S \to \Lambda$, $S \to aSb$
  or in shorthand: $S \to \Lambda | aSb$

Now, beginning the derivations. We have to start with the start symbol S, and we can either derive Λ or aSb. Obviously deriving Λ would end the production and deriving aSb would allow us to keep re-using the production rules:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow \ldots$$

Using a combination of the two production rules, we can build up a picture of what strings we can derive from the start symbol:

$$S \Rightarrow \Lambda, S \Rightarrow aSb \Rightarrow ab$$

The second string above we can turn into the following shorthand:

$$S \Rightarrow^* ab$$

Or alternatively, we can use shorthand to jump forward and yet continue the derivation:

$$S \Rightarrow^* aaaSbbb$$

This brings us to the end of the example as we can now define $L(G)$:

$$L(G) = \{\Lambda, ab, aabb, aaabbb, \ldots\}$$

Example: Longer Derivation of a String

Let $\Sigma = \{a, b, c\}$ be the set of terminal symbols and S be the only non-terminal symbol. We have four production rules:

- $S \to \Lambda$
- $S \to aS$
- $S \to bS$
- $S \to cS$

Which can alternatively be represented in Shorthand: $S \to \Lambda|aS|bS|cS$
To derive the string $aacb$ we would undergo the following derivation:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS \Rightarrow aacb\Lambda = aacb$$

Which can be shortened to $S \Rightarrow^* aacb$
Note how we started on the left and worked left-to-right. This makes this derivation a *leftmost* derivation because we produced the leftmost characters first.

2.2 Infinite Languages

As in the previous example, note how there is no bound on the length of the strings in an infinite language. Therefore there is no bound on the number of derivation steps used to derive the strings. If the grammar has n productions, then any derivation consisting of $n+1$ steps must use some production twice.

Where a language is infinite - some of the productions or sequence of productions must be used repeatedly to construct the derivations.

Example: Infinite language

Take the infinite language $\{a^n b | n \geq 0\}$ which can be described by the grammar $S \to b|aS$.
To derive the string $a^n b$, the production $S \to aS$ is used repeatedly, n times and then the derivation is stopped by using the production $S \to b$.

The production $S \to aS$ allows us to say "If S derives w, then it also derives aw".

2.3 Recursion / Indirect Recursion

A production is called recursive if its left side occurs on it's right side. For example the production $S \to aS$ is recursive.

A production $A \to \ldots$ is indirectly recursive if A derives a sentential form that contains A in two or more steps.

Example: Indirect Recursion

If the grammar contains the rules $S \to b|aA, A \to c|bS$ then both productions $S \to aA$ and $A \to bS$ are indirectly recursive:

$$S \Rightarrow aA \Rightarrow abS$$
$$A \Rightarrow bS \Rightarrow baA$$

A grammar can also be considered recursive where it contains either a recursive production or an indirectly recursive production. We can deduce from this that a grammar for an infinite language must be directly or indirectly recursive.

2.4 Constructing Grammars

Up to now, we've looked at deriving a language from a given grammar. Now we will take the inverse - be given a language and construct a grammar which derives the specified language.

Sometimes it is difficult or even impossible to write down a grammar for a given language. Unsurprisingly, a language may have more than one grammar which is correct and valid.

2.4.1 Finite Languages

If the number of strings in a language is finite, then a grammar can consist of all productions of the form $S \to w$ for each string w in the language.

Example: Finite Language

The finite language $\{a, ba\}$ can be described by the grammar:

$$S \to a|ba$$

2.4.2 Infinite Languages

To find the grammar for a language where the number of strings is infinite is a considerably bigger challenge. There is no universal method for finding a grammar for an infinite language, however the method of *combining grammars* can prove useful.

Example: Infinite Language

To find a grammar for the following simple language:

$$\{\Lambda, a, aa, \ldots, a^n, \ldots\} = \{a^n : n \in \mathbb{N}\}$$

We can use the following solution:

- We know the set of terminals: $T = \{a\}$
- We know the only non-terminal start symbol: S
- So therefore we can generate the production rules: $S \to \Lambda, S \to aS$

2.5 Combining Grammars

If we take L and M to be languages which we are able to find the grammars; then there exist simple rules for creating grammars which produce the languages $L \cup M$, $L \cdot M$, and L^*. This therefore means we can describe L and M with grammars having disjoint sets (where neither set has common elements) of non-terminals.

The combination process is started by assigning start symbols for the grammars of L and M to be A and B respectively:

$$L : A \to \ldots, \quad M : B \to \ldots$$

2.5.1 Union Rule

The union of two languages, $L \cup M$ starts with the two productions:

$$S \to A|B$$

which is followed by: the grammar rules of L (start symbol A) and then the grammar rules of M (start symbol B).

Example: Combining Grammars Using Union Rule

If we take the following language:

$$K = \{\Lambda, a, b, aa, bb, aaa, bbb, \dots, a^n, b^n, \dots\}$$

Now to find the grammar for it.
Firstly we look at it and see quite clearly there is a pattern, K is a union of the two languages:

$$L = \{a^n | n \in \mathbb{N}\} \text{ and } M = \{b^n | n \in \mathbb{N}\}$$

Therefore we can write a grammar for K as follows:

- $A \to \Lambda | aA$ (grammar for L)
- $B \to \Lambda | bB$ (grammar for M)
- $S \to A | B$ (union rule)

2.5.2 Product Rule

Much the same as the Union Rule, the product of two languages, $L \cdot M$ starts with the production:

$$S \to AB$$

Which is then followed by: the grammar rules of L (start symbol A) and then the grammar rules of M (start symbol B).

Example: Combining Grammars Using Product Rule

If we take the following language:

$$\begin{aligned} K &= \{a^m b^n | m, n \in \mathbb{N}\} \\ &= \{\Lambda, a, b, aa, ab, aaa, bb\} \end{aligned}$$

We can first find out that K is the product of two languages:

$$L = \{a^n | n \in \mathbb{N}\} \text{ and } M = \{b^n | n \in \mathbb{N}\}$$

Therefore we can write a grammar for K as follows:

- $A \to \Lambda | aA$ (grammar for L)
- $B \to \Lambda | bB$ (grammar for M)
- $S \to AB$ (product rule)

2.5.3 Closure Rule

The grammar for the closure of a language, L^*, starts with the production:

$$S \to AS | \Lambda$$

Which is followed by: the grammar rules of L (start symbol A).

Example: Grammar Closure Rule

If we take the problem that we want to construct a language, L, of all possible strings made up from zero or more occurrences of aa or bb:

$$L = \{aa, bb\}^* = M^*$$

Where $M = aa, bb$

Therefore:

$$L = \{\Lambda, aa, bb, aaaa, aabb, bbbb, bbaa, \ldots\}$$

Therefore, we can write a grammar for L as follows:

- $S \to AS|\Lambda$ (closure rule)
- $A \to aa|bb$ (grammar for $\{aa, bb\}$)

2.6 Equivalent Grammars

Grammars are not unique; a given language can have many grammars which could produce it. Grammars can be simplified down to their simplest form.

Example: Simplifying Grammars

If we take the grammar from the previous example:

$$S \to AS|\Lambda, \quad A \to aa|bb$$

We can simplify this:

- Replace the occurrence of A in $S \to AS$ by the right side of $A \to aa$ to obtain the production $S \to aaS$
- Replace A in $S \to AS$ by the right side of $A \to bb$ to obtain the production $S \to bbS$

We can therefore write the grammar in simplified form as:

$$S \to aaS|bbS|\Lambda$$

Page 3

Lecture - A3: Regular Languages

2025-10-06 14:00 Janka

3.1 What Are We Trying To Solve Here?

The problem we are trying to solve with Regular Languages is that of precision and absolute certainty - a mathematicians favourite situation.

If we take a statement: "What do we mean by a decimal number?"

We can solve this in a number of ways. One might assume *"Some digits followed maybe by a point and some more digits"* is a good description. However they would be wrong - this is imprecise and inaccurate (mathematicians nightmare).

So we can make it more precise *"Optional minus sign, any sequence of digits, followed by optional point and if so then optional sequence of digit"*. This is obviously better, and now more precise & accurate as we're acknowledging negative numbers are a thing. Although it still isn't great, there's too many words for a mathematician to approve.

So that brings us to the best option: a regular expression:

$$(- + \Lambda)DD^*(\Lambda + .D^*), D \text{ stands for a digit}$$

3.2 Regular Languages

Definitions

Regular Language A formal language that can be described by a regular expression or recognised by a finite automaton

Regular Languages are extremely useful, they are easy to recognise and describe. They provide a simple tool to solve some problems. We see regular expressions in many places within Computing - for example in pattern matching in the `grep` filter in UNIX systems or in lexical-analyser generators in breaking down the source program into logical units such as keywords, identifiers, etc.

There are four different ways we can define a regular language:

1. Languages that are *inductively* formed from combining very simple languages
2. Languages described by a *regular expression*
3. Languages produced by a grammar with a special, very restricted form
4. Languages that are accepted by some finite automaton (covered in subsequent lectures)

3.3 Defining a Regular Language with Induction

Definitions

Induction This is a process which works through the problem, situation, etc in a step-by-step way. We can inference from one step to another, for example if we know 1 is a number then 1+1 will be a number.

Defining a regular language by induction starts with the basis of a very simple language which then gets combined together in particular ways. For example, if we take L and M to be regular languages then the following languages are also regular:

$$L \cup M, L \cdot M, L^*$$

So to generalise this, for a given alphabet Σ: all regular languages over Σ can be built from combining these four in various ways by recursively using the union, product and closure operation.

Example: Regular Language Definitions

For this example, we take $\Sigma = \{a, b\}$.
This gives us four regular languages:

$$\emptyset, \{\Lambda\}, \{a\}, \{b\}$$

Ex. 1: Is the language $\{\Lambda, b\}$ regular?
Yes, it can be written as the union of two regular languages $\{\Lambda\}$ and $\{b\}$:

$$\{\Lambda\} \cup \{b\} = \{\Lambda, b\}$$

Ex. 2: Is the language $\{a, ab\}$ regular?
Yes, it can be written as the product of the two regular languages $\{a\}$ and $\{\Lambda, b\}$:

$$\{a, ab\} = \{a\} \cdot \{\Lambda, b\} = \{a\} \cdot (\{\Lambda\} \cup \{b\})$$

Ex. 3: Is the language $\{\Lambda, b, bb, \ldots, b^n, \ldots\}$ regular?
Yes, it is the closure of the regular language $\{b^*\}$:

$$\{b\}^* = \{\Lambda, b, bb, \ldots, b^n, \ldots\}$$

Ex. 4: Is the language $\{a, ab, abb, \ldots, ab^n, \ldots\}$ regular?
Yes, we can construct it:

$$\{a, ab, abb, \ldots, ab^n, \ldots\} = \{a\} \cdot \{\Lambda, b, bb, \ldots, b^n, \ldots\} = \{a\} \cdot \{b\}^*$$

Ex. 5: Is the language $\{b, aba, aabbaa, \ldots, a^n ba^n, \ldots\}$ regular?
No. This cannot be regular because we have now way to ensure that the two sets of a are both repeated n times.

There are additional examples in the Lecture A3 slides on Moodle.

So what we've learnt from the above is that regular languages can be finite or infinite, and that they cannot have the same symbol repeated in two different places the same number of repetitions.

3.4 Defining a Regular Language with Regular Expressions

Definitions

Regular Expression A sequence of characters that define a specific search pattern for matching text

In our use case, a *Regular Expression* is a shorthand way of showing how a regular language is built from the bases set of regular languages. It uses symbols which are nearly identical to those used to construct the language. Any given regular expression has a language closely associated with it.

For each regular expression E, there is a regular language $L(E)$.

Much like the languages they represent, a regular expression can be inductively manipulated to form new regular expressions. For example if we take R and E as regular expressions then the following are also regular:

$$(R), R + E, R \cdot E, R^*$$

Example: Regular Expresssions

If we take the alphabet to be $\Sigma = \{a, b\}$ then listed below are some of the infinitely many regular expressions:

$$\Lambda, \emptyset, a, b$$

$$\Lambda + b, b^*, a + (b \cdot a), (a + b) \cdot a, a \cdot b^*, a^* + b^*$$

Much like maths, we have an order of operations to help us understand how to interpret a given regular expression. This goes, evaluated first to last: ( ), $*$, $\cdot$, $+$

It's worth noting that the $\cdot$ symbol is often dropped so instead of writing $a + b \cdot a^*$, you would write $a + ba^*$; in it's bracketed form - this would be $(a + (b \cdot (a^*)))$.

The symbols of the regular expressions are distinct from those of the languages, as can be seen in the following table. The language will always be either an empty set, or a set.

Regular Expression	Language
$\emptyset$	$L(\emptyset) = \emptyset$
Λ	$L(\Lambda) = \{\Lambda\}$
a	$L(a) = \{a\}$

Table 3.1: Comparison of Regular Expression syntax and Language syntax

There are two binary operations on regular expressions ($+$ and $\cdot$) and one unary operator ($*$). These are closely associated with the union ($+$), product ($\cdot$) and closure ($*$) operations on the corresponding languages.

Example: Regular Language Operations

The regular expression $a + bc^*$ is effectively shorthand for the regular language:

$$\{a\} \cup (\{b\} \cdot (\{c\}^*))$$

Example: Translating a Regular Expression into a Language

If we take the regular expression $a + bc^*$, we can find it's language:

$$\begin{aligned}
L(a + bc^*) &= L(a) \cup L(bc^*) \\
&= L(a) \cup (L(b) \cdot L(c^*)) \\
&= L(a) \cup (L(b) \cdot L(c)^*) \\
&= \{a\} \cup (\{b\} \cdot \{c\}^*) \\
&= \{a\} \cup (\{b\} \cdot \{\Lambda, c, c^2, \ldots, c^n, \ldots\}) \\
&= \{a\} \cup \{b, bc, bc^2, \ldots, bc^n, \ldots\} \\
&= \{a, b, bc, bc^2, \ldots, bc^n, \ldots\}
\end{aligned}$$

Example: Translating from Language to Regular Expression

If we take the regular language:

$$\{\Lambda, a, b, ab, abb, abbb, \ldots, ab^n, \ldots\}$$

We can represent with a regular expression:

$$\Lambda + b + ab^*$$

We've used *union* not *product* because the language doesn't include a leading b; there are three options for the strings structure:

Λ the empty string
b a singular b on it's own
ab^* a single a followed by zero or more b

Regular Expressions may not be unique, in that two or more different regular expressions may represent the same languages. For example the regular expressions $a + b$ and $b + a$ are different, but they both represent the same language:

$$L(a + b) = L(b + a) = \{a, b\}$$

We can say that regular expressions R and E are equal if their languages are the same (i.e. $L(R) = L(E)$), and we denote this equality in the familiar way $R = E$.

There are many general equalities for regular expressions. All the properties hold for any regular expressions R, E, F and can be verified by using properties of languages and sets.

Additive (+) properties

$$\begin{aligned}
R + E &= E + R \\
R + \emptyset &= \emptyset + R = R \\
R + R &= R \\
(R + E) + F &= R + (E + F)
\end{aligned}$$

Product (·) properties

$$\begin{aligned}
R\emptyset &= \emptyset R = \emptyset \\
R\Lambda &= \Lambda R = R \\
(RE)F &= R(EF)
\end{aligned}$$

Distributive Properties

$$\begin{aligned}
R(E + F) &= RE + RF \\
(R + E)F &= RF + EF
\end{aligned}$$

Closure Properties

$$\begin{aligned}
\emptyset^* &= \Lambda^* = \Lambda \\
R^* &= R^*R^* = (R^*)^* = R + R^* \\
R^* &= \Lambda + RR^* = (\Lambda + R)R^* \\
RR^* &= R^*R \\
R(ER)^* &= (RE)^*R \\
(R + E)^* &= (R^*E^*)^* = (R^* + E^*)^* = R^*(ER^*)^*
\end{aligned}$$

We can use a combination of these properties to simplify regular expressions and prove equivalences.

Example: Regular Expression Equivalence

Show that: $\Lambda + ab + abab(ab)^* = (ab)^*$ Using the above properties.

$$\begin{aligned}
\Lambda + ab + abab(ab)^* = (ab)^* &= \Lambda + ab(\Lambda + ab(ab)^*) \\
&= \Lambda + ab((ab)^*) \quad (\text{Using } R^* = \Lambda + RR^*) \\
&= \Lambda + ab(ab)^* \\
&= (ab)^* \quad (\text{Using } R^* = \Lambda + RR^* \text{ again})
\end{aligned}$$

3.5 Defining a Regular Language using Regular Grammars

Definitions

Regular Grammar A grammar where each production takes one of the following restricted forms:

$$\begin{aligned}
&B \to \Lambda,\ B \to w, \\
&B \to A, \\
&B \to wA
\end{aligned}$$

Where A, B are non-terminals and w is a non-empty string of terminals.

There are two regulations all regular grammars must adhere to:

- Only one non-terminal can appear on the right hand side of a production
- Non-terminals must appear on the right end of the right hand side

Therefore, $A \to aBc$ and $S \to TU$ are not part of a regular grammar. The production $A \to abcA$ is, however. Obviously things like $A \to aB|cC$ are allowed because they are two separate productions.

For any given regular language, we can find a regular grammar which will produce it. However there may also be other non-regular grammars which also produce it.

Example: Regular and Irregular Grammars

If we take the regular language a^*b^*, we can see it has a regular grammar and an irregular grammar.

Irregular Grammar

$$\begin{aligned}
S &\to AB \\
A &\to \Lambda | aA \\
B &\to \Lambda | Bb
\end{aligned}$$

Regular Grammar

$$\begin{aligned}
S &\to \Lambda | aS | A \\
A &\to \Lambda | bA
\end{aligned}$$

Page 4

Lecture - A4: Finite Automata

2025-10-06 15:00 Janka

This topic will continue into lecture A5 (next Monday).

4.1 Models of Computation

In this module we'll study different models of computation. These are theoretical ways of representing the computation which is going on within the computer for a given scenario. Examples include *finite automata*, *push-down automata* and *Turing machines*.

All these models have an *input tape*. This is a continuous input string which is divided up into single string segments. The models can either accept or reject the input strings based on their rules. The set of all accepted strings over the alphabet is the language recognised by the model.

They have different types of memory - some may have finite and others infinite. Some models may have additional features.

4.2 Finite Automata: An Introduction

The most basic model of a computer is the *Finite Automata* (FA). These have three components:

- an *input tape* which contains an input string over Σ
- a *head* which reads the input string one symbol at a time
- some *memory* which is a finite set of Q states. The FA is always only in one state, called a *current state* of the automaton

The *program* of the FA defines how the symbols that are read change the current program.

Finite Automata are commonly represented as a transition graph (directed graph, cue flashbacks to DMAFP) because they are simpler to interpret than the formal definitions, which we will cover later.

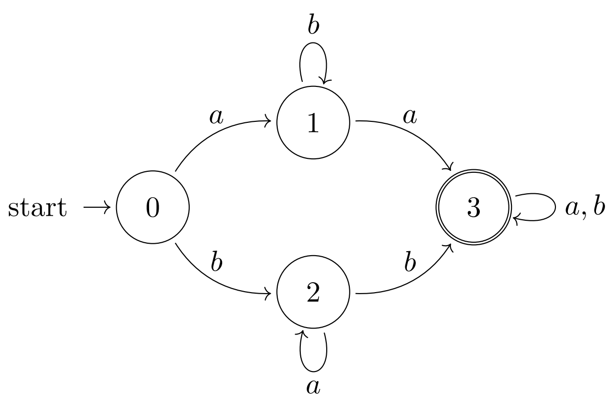

Figure 4.1: Example of Finite Automata

All *finite automata* will have one *initial* state and at least one *final state* (denoted by a double circle).

FAs work by starting in the initial state (0) and as we read off symbols in the string we move from state-to-state (vertex-to-vertex). If after reading the entire input string, the automaton is in the final state - the input string is accepted; if the automaton is not in the final state - the input string is rejected.

To define the function of a FA in mathematical terms - they read a finite-length input string over Σ, one symbol at a time. A FA is always in a *state*, from the set of states Q. They begin in a designated initial (start) state, then on reading a symbol - the state changes which is called a transition. The new state depends on the current state and the symbol read in. There is no option to re-read the input symbols or to write them anywhere. At the end of the string, the machine either accepts the string if and only if its state is one of the final state, otherwise it gets rejected. The language of the automaton is the set of strings it accepts.

Example: Finite Automata Processing

Looking at the Automata in Figure 4.1, we can take the example input *abbbba*.
This would start in state 0, and travel to state 1, accepting the initial a. The automata then loops around from state 1 to state 1 accepting the b, which is repeated 4 times in total. The automata then takes the final a and transitions from state 1 to 3. As we have processed all the input string and we are in the final state - the input string is accepted.
Below is a representation of the transitions taken by the automata to process the input string:

$$0 \to^a \ 1 \to^b \ 1 \to 1^b \ 1 \to 1^b \ 1 \to 1^b \ 1 \to 3^a$$

4.2.1 State Transition Functions

As much as pretty pictures of Finite Automata are all well and good - there is a second way to represent the transitions: using *transition functions*.

Transition functions take the form:

$$T : Q \times \Sigma \to Q$$

Where Q is the set of states and Σ is the alphabet. We can see below an abstracted FA showing two states i and j, with a single symbol a:

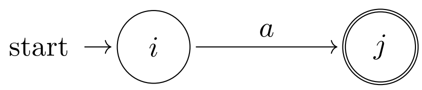

Figure 4.2: Abstracted FA with two states

In the above Finite Automata (Figure 4.2) we can see how the transition function behaves. It is represented by $T(i, a) = j$, where $i, j \in Q$ and $a \in \Sigma$.

Example: State Transition Functions

If we take a more complex Finite Automata:

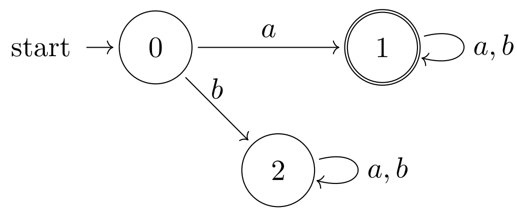

Figure 4.3: Example Finite Automata for Transition Functions

We know it has a set of states: $Q = \{0, 1, 2\}$; a start state: 0; and some final state(s): 1.
We can take the transition function and see the possible transitions over $\Sigma = \{a, b\}$:

$$T(0, a) = 1,\ T(0, b) = 2,$$
$$T(1, a) = T(1, b) = 1$$
$$T(2, a) = T(2, b) = 2$$

4.3 Deterministic Finite Automata

Definitions

Deterministic Finite Automata a DFA over a finite alphabet Σ is a finite directed graph with the property that each node emits one labelled edge for each distinct element of Σ

Except, hang on - isn't that what we've just seen so far. Yes, all the examples explored in this lecture so far have been DFAs; as there is exactly one option of transition for every state and every symbol, with every node in the graph having exactly one edge coming out for each possible input symbol.

We can define DFA more formally in that a DFA *accepts* a string w over Σ^* if there is a path from the start state to a final state such that w is the concatenation of the edges of the path; otherwise the DFA *rejects* w.

We also need to know that the set of all strings over Σ accepted by a DFA M is called the language of M and is denoted as $L(M)$.

For any regular language, a DFA can be found which recognises it. This will be proved in the next lecture.

Example: Constructing a DFA for a given Regular Expression

If we take the following regular expression:

$$(a + b)^*abb \text{ over the alphabet } \Sigma = \{a, b\}$$

We can make an observation: the language is the set of strings that begin with anything but must end with the string *abb* therefore we're looking for strings which have a particular patten to them. This method could be extended if we had a bigger alphabet, for example if we were looking for all strings ending in .tex, or .pdf.
The challenge with this regular expression is that we won't know when the string will end. For example, the string could be *abb*, or *ababababab b*. So to get around this, we will keep track of the last three symbols we've seen:

- If in state 1: the last character was a
- If in state 2: the last two symbols were ab
- If in state 3: the last three wee abb

With this in mind, we can now construct the DFA.

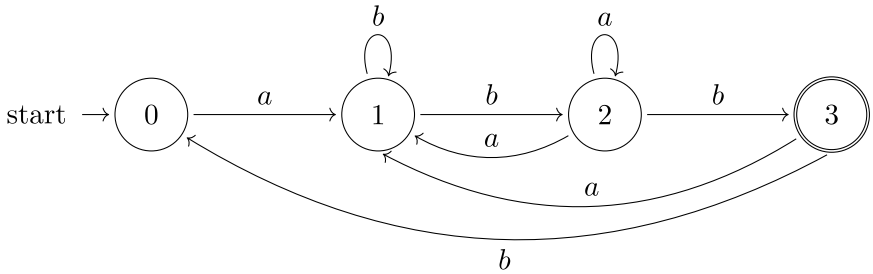

Figure 4.4: DFA constructed from Regular Expression

There are additional examples of DFA construction for a given regular expression in the slides available on Moodle.

4.4 Non-Deterministic Finite Automata

As we know with *Deterministic* Finite Automata - we know exactly which state it is in and the path it took to get there for any given input string. To take the inverse of this - *Non-Deterministic* Finite Automata (NFA) may have more than one option we can follow with the same input character, or there may be no option for a given input character. A NFA accepts and rejects strings in the same way as a DFA: accepting any string which ends up in its final state and rejecting everything else.

A NFA over an alphabet Σ is a finite transition graph with each node having zero or more edges. Each edge is labelled with either a letter from Σ or Λ. Multiple edges may go from the same node with the same label, and some letters may not have an edge associated with them - strings following such paths are rejected.

If an edge is labelled with the empty string Λ, then we can move to the next state (along the edge) without consuming an input letter - effectively we could be in either state and so the possible paths could branch. If there are two edges with the same label from one node, we can move along any of them.

Example: Construct a NFA for a given Regular Expression

If we take the regular expression $ab + a^*a$. We can draw a NFA to recognise the language of it.

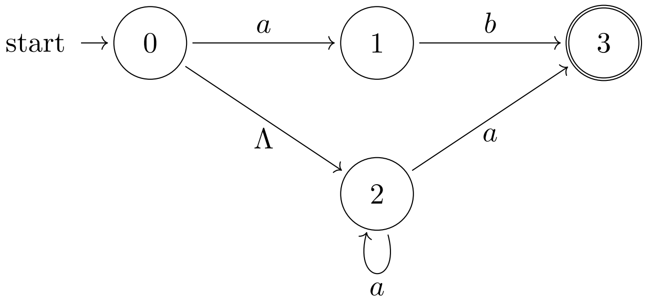

Figure 4.5: Example Non-Deterministic Finite Automata

In this NFA, we see that the "upper" path corresponds to ab and the "lower" path to a^*a. We know this is a NFA because it has a Λ edge and two $a-$edge from state 2.

Due to the non-deterministic nature of a NFA, the output of the transition functions are sets of states, $T : Q \times \Sigma \to P(Q)$.

Example: NFA Transition Functions

For example, if there are no edges from state k labelled with a, we'll write:

$$T(k, a) = \emptyset$$

If there are three edges from state k all labelled with a going to states i, j, and k, we'll write:

$$T(k, a) = \{i, j, k\}$$

Looking back at Figure 4.5 from the previous example, we can see there are four states 0, 1, 2, 3; where 0 is the starting state and 3 is the final state. From here we can see the transition functions:

$$\begin{aligned} T(0, a) &= \{1\} \\ T(0, \Lambda) &= \{2\} \\ T(1, b) &= \{3\} \\ T(3, a) &= \{2, 3\} \end{aligned}$$

4.5 DFA vs. NFA

All digital computers are deterministic. The usual mechanism for deterministic computers is to try one particular path and then to backtrack to the last decision point if that path proves to be poor. Parallel computers make non-determinism almost realisable; for example, we can let each process make a random choice at each branch point thereby exploring many possible trees.

Generally speaking, NFAs are easier to construct and tend to be simpler with fewer states, for a given regular expression to recognise. However, DFAs are easier to operate as the path followed is always unique. Given that they recognise the same language, one is always able to find a DFA which recognises the language of a given NFA. DFAs are a subset of NFAs, so we only need to show that we can map any NFA into a DFA.

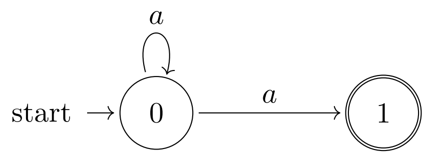

Figure 4.6: Example NFA for a^*a

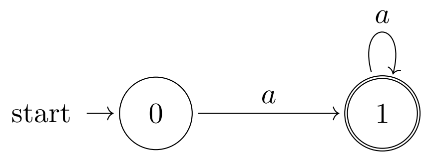

Figure 4.7: Example DFA for $a^*a = aa^*$

4.6 Finding an Equivalent DFA for a given NFA

We can prove the equivalence of NFAs and DFAs by showing how for any NFA by constructing a DFA which recognising the same language. Generally the DFA will have more possible states than the NFA; if the NFA has n states then the DFA could have as many as 2^n states.

Example: Converting a NFA to a DFA

If we take the following NFA which recognises the language $(a + b)^*ab$ over the alphabet $\{a, b\}$

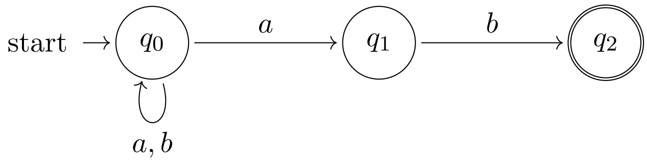

Figure 4.8: NFA To Be Converted

Step 1
Begin in the NFA stat state; if it is connected to any others by Λ, the DFA start state could be a set of states.

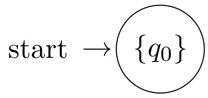

Figure 4.9: Start symbol of DFA

Step 2
For each symbol - determine the set of possible NFA states you could be in after reading it. This set is a label for a new DFA state and is connected to the start by that symbol. In our example - the start state is q_0, but following an a you could be in q_0 or q_1; following a b you could only be in state q_0.

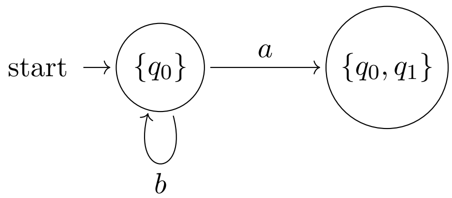

Figure 4.10: First step of converting NFA to DFA

Step 3
Repeat step 2 for each new DFA state, exploring the possible results for each symbol until the system is closed.

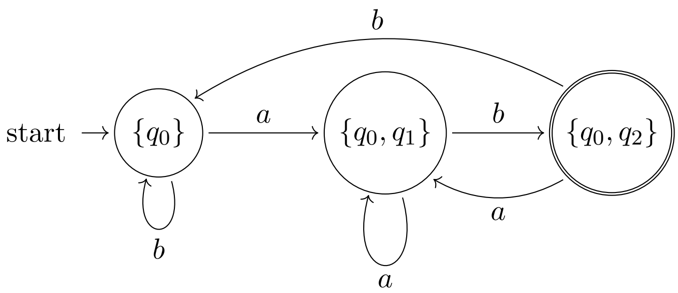

Figure 4.11: DFA showing all valid states

The final state of the DFA are those that include an NFA final state in the set.
If there is no transition for a state / a symbol in the NFA (non-acceptance of the string), create a new state in the DFA labelled $\emptyset$ and add loops for all symbols (a non-final trap state).

4.6.1 Trap States

Definitions

Trap State A state in which the machine cannot reach any final or accepting state.

Trap States are needed in DFAs because to satisfy the requirement to be a DFA - every state must have an outgoing transition for every symbol in the alphabet. This is not an issue in NFAs because the Automaton assumes that if it can't find a suitable transition to use - the input string is invalid.

Example: Trap States in Action

If we take the regular expression from our previous example, $(a+b)^*ab$ and expand the alphabet used to be $\{a, b, c\}$. This presents a problem as the input could contain c, but there's no suitable transitions the DFA can take for such a letter. We use a *trap state* to catch this pesky c.
We can add a trap state to the DFA we created in the previous example.

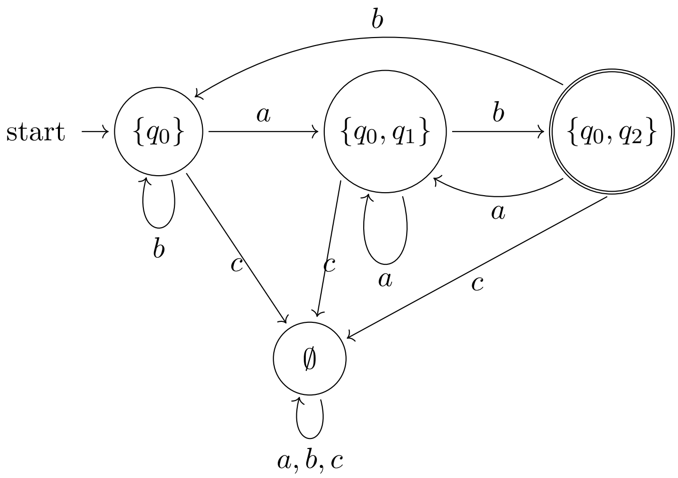

Figure 4.12: DFA showing a Trap State