BSc (Hons) Computer Science

University of Portsmouth
Third Year

Theoretical Computer Science

M21276
Semester 1

Hugh Baldwin

hugh.baldwin@myport.ac.uk

Contents

Lecture - Induction Lecture

13:00   25/09/24   Janka Chlebikova

Contact Time

- Lectures

    - Delivered by Janka

    - Slides and videos from Covid times available weekly on Moodle

- Tutorials

    - Delivered by either Janka or Dr Paolo Serafino

    - Worksheets available with solutions weekly on Moodle

    - 5 Groups

    - Work through the worksheet solutions

    - Kahoot quiz for revision

    - 'Tutorial 0' available as revision of concepts from DMAFP needed for this module

Assessments

- 50%, 90 minute In-Class Test covering Part A - November 20th, 2pm

- 50%, 90 minute Exam covering Part B (Only) - TB1 Assessment Period (January)

- Deferred assessment covers both Part A & B, so will be harder than the two separate exams

There are two past papers (paper-based, but still relevant to the computer-based test) available for each of the assessments. As with DMAFP, the solutions are supplied in the form of a crowd-sourced document that Janka will check over before the exam.

Resources

Lecture slides, videos and tutorial papers are released every week. All of the books on the reading list are available online through the links on Moodle.

Lecture - A1: Introduction to Languages

13:00   03/10/24   Janka Chlebikova

Languages

In this context, a **language** is a set of symbols which can be combined to create a list of acceptable **strings**. There are then also rules which tell us how to combine these strings together, known as **grammars**. The combination of an alphabet, list of valid strings and grammars is known as a language. There will be a formal definition later on.

> **Definition**
>
> An **Alphabet** is a finite, non-empty set of symbols.

For example, in the English language we would define the alphabet, A, as $A = \{a, b, c, d, \ldots, x, y, z\}$. These symbols can then be combined to create a **string**.

> **Definition**
>
> A **String** is a finite sequence of symbols from the alphabet of a language.

With the alphabet A, we could have strings such as 'cat', 'dog', '$antidisestablishmentarianism$', etc *over* the alphabet. A string with no symbols, and therefore a length of zero, is known as the **empty string**, and is denoted by Λ (capital lambda).

> **Definition**
>
> A **Language** over an alphabet (e.g. English over A) is a set of strings – including Λ – made up of symbols from A which are considered 'valid'. They could be valid as per a set of rules, or could be arbitrary as in most spoken languages.

If we have an alphabet, Σ, then Σ^* denotes the infinite set of all strings made up of symbols in Σ – including Λ. Therefore, a language over Σ is any subset of Σ^*.

> **Example**
>
> If $\Sigma = \{a, b\}$, then $\Sigma^* = \{\Lambda, a, b, aa, ab, ba, bb, \ldots\}$
> There are many languages which could be defined over this alphabet, but a few simple one are
> - $\emptyset$ (The empty set)
> - $\{\Lambda\}$ (The set containing an empty string)
> - $\{a\}$ (The set containing a)
> - The infinite set $\Sigma^* = \{\Lambda, a, aa, aaa, \ldots\}$

Combining Languages

There are three common ways of combining two languages to create a new language – union, intersection and product.

Union and Intersection

Since languages are sets of strings, they can be combined as you usually would for any other set with the usual operations, union and intersection.

> **Example**
>
> If $L = \{aa, bb, cc\}$ and $M = \{cc, dd, ee\}$, then
> $L \cap M = \{cc\}$, and
> $L \cup M = \{aa, bb, cc, dd, ee\}$

Product

To combine two languages L and M, we form the set of all **concatenations** of strings in L with strings In M. This is known as the **product** of the two languages.

> **Definition**
>
> To **Concatenate** two strings is to juxtapose them such that a new string is made by appending the second string to the end of the first.

> **Example**
>
> If we concatenate the strings ab and ba, the result would be $abba$.
> This can be represented using the function cat, such as $\mathrm{cat}(ab, ba) = abba$

With this definition of concatenation, we can say that the product of two languages, L and M would be $L \cdot M$, where $L \cdot M = \{\mathrm{cat}(s, t) : s \in L \text{ and } t \in M\}$

> **Example**
>
> If $L = \{a, b, c\}$ and $M = \{c, d, e\}$, then the product of the two languages, $L \cdot M$, is the language
> $L \cdot M = \{ac, ad, ae, bc, bd, be, cc, cd, ce\}$

Further on Products

It is simple to see that for any language, L, the following simple properties are true:

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

Commutativity

Aside from the above properties, the product of any two languages is **not** commutative, and therefore

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

> **Example**
>
> If $L = \{a, b\}$ and $M = \{b, c\}$, then the product $L \cdot M$ is the language
>
> $$L \cdot M = \{ab, ac, bb, bc\}$$
>
> but the product $M \cdot L$ is the language
>
> $$M \cdot L = \{ba, bb, ca, cb\}$$
>
> which are clearly not the same language, as the only common string is bb

Associativity

However, the product of any two languages is associative. This means that if we had any three languages, L, M and N, then

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

> **Example**
>
> If we have the three languages $L = \{a, b\}$, $M = \{b, c\}$ and $N = \{c, d\}$, then
>
> $$\begin{aligned} L \cdot (M \cdot N) &= L \cdot \{bc, bd, cc, cd\} \\ &= \{abc, abd, acc, acd, bbc, bbd, bcc, bcd\} \end{aligned}$$
>
> which is the same as
>
> $$\begin{aligned} (L \cdot M) \cdot N &= \{ab, ac, bb, bc\} \cdot N \\ &= \{abc, abd, acc, acd, bbc, bbd, bcc, bcd\} \end{aligned}$$

Powers of Languages

If we have the language L, then the product $L \cdot L$ of the language is denoted by L^2. The product L^n for every $n \in \mathbb{N}$ is defined as

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

> **Example**
>
> If $L = \{a, b\}$ then the first few powers of L are
>
> $$\begin{aligned} L^0 &= \{\Lambda\} \\ L^1 &= L = \{a, b\} \\ L^2 &= L \cdot L = \{aa, ab, ba, bb\} \\ L^3 &= L \cdot L^2 = \{aaa, aab, aba, abb, baa, bab, bba, bbb\} \end{aligned}$$

The Closure of a Language

If L is a language over Σ, then the **closure** of L is the language denoted by L^*, and the **positive closure** is language denoted by L^+.

> **Definition**
>
> The **Closure** of the language L, L^* is defined as
>
> $$L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$$
>
> and so, if $L = \{a\}$ then
>
> $$L^* = \{\Lambda\} \cup \{a\} \cup \{aa\} \cup \{aaa\} \cup \ldots$$
> $$= \{\Lambda, a, aa, aaa, \ldots\}$$

> **Definition**
>
> The **Positive Closure** the language L, L^+ is defined as
>
> $$L^+ = L^1 \cup L^2 \cup L^3 \cup L^4 \cup \ldots$$
>
> and so, if $L = \{a\}$ then
>
> $$L^+ = \{a\} \cup \{aa\} \cup \{aaa\} \cup \{aaaa\} \cup \ldots$$
> $$= \{a, aa, aaa, aaaa, \ldots\}$$

It then follows that $L^* = L^+ \cup \{\Lambda\}$, but it's not necessarily true that $L^+ = L^* - \{\Lambda\}$.

> **Example**
>
> If our alphabet is $\Sigma = \{a\}$ and our language is $L = \{\Lambda, a\}$, then
>
> $$L^+ = L^*$$

Properties of Closures

Based upon these definitions, you can derive some interesting properties of closures.

> **Example**
>
> If L and M are languages over the alphabet Σ, then
>
> $$\{\Lambda\}^* = \emptyset^* = \{\Lambda\}$$
> $$L^* = L^* \cdot L^* = (L^*)^*$$
> $$\Lambda \in L \text{ if and only if } L^+ = L^*$$
> $$(L^* \cdot M^*)^* = (L^* \cup M^*)^* = (L \cup M)^*$$
> $$L \cdot (M \cdot L)^* = (L \cdot M)^* \cdot L$$
>
> *These will be explored more during the tutorial session for this week*

The Closure of an Alphabet

Going back to the definition of Σ^* of the alphabet Σ, it lines up perfectly with the definition of a closure such that Σ^* is the set of all strings over Σ. This means that there is a nice way to represent Σ^* as follows

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots$$

From this, we can also see that Σ^k denotes the set of all strings over Σ whose length is k.

Lecture - A2: Grammars

13:00	03/10/24	Janka Chlebikova

As discussed previously, you can define a language over an alphabet as a set of arbitrary strings that are considered 'valid', but you can also define a language using a **grammar**.

> **Definition**
>
> A **Grammar** is a set of rules used to define a language over an alphabet, by specifying the structure of valid strings in said language.

To describe the grammar of a language, two sets of symbols (alphabets) are required, terminals and non-terminals

> **Definition**
>
> **Terminal** symbols are those from which the actual strings which make up a language are derived, like the symbols discussed previously for a manually specified language. In the case of a spoken language, these would usually be lowercase letters, such as the latin alphabet used in English.

> **Definition**
>
> **Non-Terminal** symbols are 'temporary' symbols (fully disjoint from the set of terminal symbols) used to define the grammar's replacement rules. These must all be replaced by terminals before a production can be considered a valid string in the language. These are usually represented by uppercase letters, or letters from an alphabet other than that of the language.

Productions

Further to this, a grammar for the language L over Σ would typically consist of a set of productions (grammar rules) which allow you to produce the set of strings which make up L.

> **Definition**
>
> A **Production** is a rule which allows you to produce a string for a language. They are typically in the form
>
> $$\alpha \to \beta$$
>
> where α is a string of symbols from the set of terminals (Σ) and β is a string of symbols from the set of non-terminals.

You can read these rules in several ways – $\alpha \to \beta$ could be read as;

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

A Formal Definition

Formally, a grammar is defined by the following properties

1. An alphabet T of terminal symbols (This is the same as the alphabet of the language defined by this grammar)

2. An alphabet N of non-terminal symbols

3. A specific non-terminal symbol known as the **start symbol**, which is often S

4. A finite set of productions in the form $\alpha \to \beta$ where α and β are strings over the alphabet $N \cup T$

Every grammar must have the special non-terminal symbol known as a start symbol, and it must also have at least one production in which the left side consists of only the start symbol.

> **Example**
>
> Let G be a grammar defined by
> - The set of terminals $T = \{a, b\}$
> - A single non-terminal being the start symbol S
> - The set of productions
>
> $$S \to \Lambda,\ S \to aSb$$
>
> or for shorthand,
>
> $$S \to \Lambda \mid aSb$$
>
> To get the full set of strings which are valid for this grammar, we need to perform a **derivation**

Derivations

Starting from any production where the left side consists of only the start symbol, we can go through a step by step process to generate all strings belonging to the language described by a grammar.

> **Example**
>
> Using the grammar G from the previous example, we could start with either of the two productions, since they both have only the start symbol S on the left-hand side. This means that the first step of our process would give us Λ and aSb.

In this example, the string aSb is a **sentential form** of the terminals $\{a, b\}$ and non-terminal S. From this point, to extend or generate any other strings, we need to perform a derivation.

> **Definition**
>
> If x and y are sentential forms, and $\alpha \to \beta$ is a production, then to replace α by β in $x\alpha y$ is to **Derive** a new string. This is denoted by writing
>
> $$x\alpha y \Rightarrow x\beta y$$

> **Example**
>
> Going back to the grammar G again, since it contains the production $S \rightarrow aSb$, we could take our first step to generate the string aSb, then we can go another step further and derive $aaSbb$ from aSb, which means
> $$aSb \Rightarrow aaSbb$$
> Since we can use this production again and again, we could derive that
> $$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow \dots$$

To represent this multi-step process, we have the following symbols to show how the derivation was constructed

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

> **Example**
>
> Using the grammar G, we can show multiple derivations using these new symbols
> $$S \Rightarrow \Lambda, S \Rightarrow aSb, \therefore S \Rightarrow^* ab$$
> $$S \Rightarrow^* aaabbb$$
> $$S \Rightarrow^* aaaaaaSbbbbbb$$

What is $L(G)$?

> **Definition**
>
> If G is a grammar with the start symbol S and the set of terminals T, then the language generated by G is the set
> $$L(G) = \{s \mid s \in T^* \text{ and } S \Rightarrow^+ s\}$$
> That is to say that it's the set of all strings containing only terminal symbols which can be derived from the start symbol with one or more steps.

> **Example**
>
> With the grammar G,
> $$L(G) = \{\Lambda, ab, aabb, aaabbb, aaaabbbb, \dots\}$$

Infinite Languages

Within an infinite language, there is no limit on the length of strings, and therefore also no limit on the number of steps needed to derive a string. If the grammar of a language has n productions, then any derivation with $> n + 1$ steps must use at least one production twice. Therefore, if a language is infinite, then a production or sequence of productions must be used repeatedly to construct the derivations of the grammar.

> **Example**
>
> The infinite language $\{a^n b \mid n \geq 0\}$ can be described by a grammar with the production $S \to b \mid aS$. To derive the string $a^n b$, you would apply the production $S \to aS$ n times over and then finish the derivation with the production $S \to b$.

With the production $S \to aS$, we can also say that "If S derives w, then it also derives aw"

Recursion and Indirect Recursion

> **Definition**
>
> A production is **Recursive** if its entire left side is contained within the right side.

> **Example**
>
> The production $S \to aSb$ is recursive, since the sentential form it produces contains itself.

> **Definition**
>
> A production is **Indirectly Recursive** if it is possible to derive a sentential form in two or more steps which contains the entire left side of the production.

> **Example**
>
> If a grammar contains the rules $S \to b \mid aA$ and $A \to c \mid bS$, then both productions are indirectly recursive, since
>
> $$S \Rightarrow aA \Rightarrow abS$$
> $$A \Rightarrow bS \Rightarrow baA$$

Recursive Grammars

> **Definition**
>
> A grammar is **Recursive** if it contains either a recursive or indirectly recursive production. *In order for a grammar to be infinite, it must first be recursive.*

Constructing Grammars

We've seen previously how to derive a language from the definition of a grammar, but we also need to be able to construct a grammar to produce a language. It is often very difficult and sometimes impossible to write down a grammar for a given language, and there are often multiple grammars which form the exact same language.

Finite Languages

In the case of a finite language, it is always possible to find a grammar which produces the language. This is because it is possible to simply make a grammar which contains a production for every string of the language, where $S \to w$ for every string w in the language.

> ### Example
>
> The finite language $\{a, b, c, abc\}$ over the alphabet $\{a, b, c\}$ can be described by the grammar
>
> $$S \to a|b|c|abc$$

Infinite Languages

However, in the case of an infinite language, there is no universal method that will always produce a valid grammar. This means it is much harder to find a valid grammar, and often requires **combining grammars**.

> ### Example
>
> A simple example of an infinite language would be $\{\Lambda, a, aa, aaa, \ldots\}$ which we can also write in the form $\{a^n : n \in \mathbb{N}\}$. We can then see that one grammar which describes this language is as follows;
> - The set of terminals, $T = \{a\}$
> - The non-terminal start symbol, S
> - The productions $S \to \Lambda, S \to aS$

Combining Grammars

Suppose we have two languages, L and M for which there are known grammars. There are a few simple rules we can use to create the grammars which describe the languages $L \cup M$, $L \cdot M$ and L^*.

To create this 'composite' grammar, we can describe L and M with disjoint sets of non-terminals. If we have the start symbols of L and M as A and B respectively, i.e. $L : A \to \ldots$, $M : B \to \ldots$, we can then combine these as productions in another grammar to create the language.

Unions

If we need to produce the union of two languages, $L \cup M$, we start with the two productions $S \to A \mid B$, followed by the productions which make up L and M, using A and B as their respective start symbols.

> ### Example
>
> If we wanted to write the grammar that produces the language
>
> $$K = \{\Lambda, a, b, aa, bb, aaa, bbb, \ldots, a^n, b^n\}$$
>
> we can realise that K is the union of two different languages,
>
> $$L = \{a^n \mid n \in \mathbb{N}\}$$
> $$M = \{b^n \mid n \in \mathbb{N}\}$$
>
> and so we can write a grammar for K as
> - $S \to A \mid B$ (Union rule)
> - $A \to \Lambda \mid aA$ (Ls grammar)
> - $B \to \Lambda \mid bB$ (Ms grammar)

Products

Similarly, if we need to produce the the product of two languages, $L \cdot M$, we start with the production $S \to AB$, followed by the productions which make up L and M, using A and B as their respective start symbols.

> **Example**
>
> If we wanted to write the grammar that produces the language
>
> $$K = \{\Lambda, a, b, aa, ab, aaa, bb, \ldots\}$$
>
> we can realise that K is the product of the two languages,
>
> $$L = \{a^n \mid n \in \mathbb{N}\}$$
> $$M = \{b^n \mid n \in \mathbb{N}\}$$
>
> and so we can write a grammar for K as
> - $S \rightarrow AB$ (Product rule)
> - $A \rightarrow \Lambda \mid aA$ (Ls grammar)
> - $B \rightarrow \Lambda \mid bB$ (Ms grammar)

Closures

And finally, if we need to produce the closure of a language, L^*, we start with the production $S \rightarrow AS \mid \Lambda$ followed by the productions which make up L, using A as its start symbol.

> **Example**
>
> If we wanted to write the grammar that produces the language L^* where
>
> $$L = \{aa, bb\}$$
>
> and so
>
> $$L^* = \{\Lambda, aa, bb, aaaa, aabb, bbbb, bbaa, \ldots\}$$
>
> we can write the grammar for L^* as
> - $S \rightarrow AS \mid \Lambda$ (Closure rule)
> - $A \rightarrow aa \mid bb$ (Ls grammar)

Equivalent Grammars

Any given language could have many different grammars which produce the exact same thing. This means that grammars are not unique. We can also use this to simplify grammars.

> **Example**
>
> Take the grammar from the previous example,
>
> $$S \rightarrow AS \mid \Lambda$$
> $$A \rightarrow aa \mid bb$$
>
> If we replace A in S with the right side of A, aa, we get the production $S \rightarrow aaS$. We can then do the same with the other production to get $S \rightarrow bbS$. Therefore, we can write this grammar in a simplified form as
>
> $$S \rightarrow aaS \mid bbS \mid \Lambda$$

Lecture - A3: Regular Languages

13:00   10/10/24   Janka Chlebikova

A regular language is a simple sort of language that meets a few basic requirements. They are often used for pattern matching and in lexical analysers.

Definition

There are several ways of describing a regular language
- Languages inductively formed by combining simple languages
- Languages described by a regular expression
- Languages produced by a grammar with a special and restricted form
- Languages that can be accepted by a finite automata

Inductively Formed Regular Languages

We start with a very simple language or set thereof and build more complex ones by combining them in particular ways–

- **Basis** - $\emptyset$, $\{\Lambda\}$ and $\{a\}$ are all regular languages for all $a \in \Sigma$

- **Induction** - If L and M are regular languages, then $L \cup M$, $L \cdot M$ and L^* are also regular languages

The basis of this definition gives us the following four regular languages over the alphabet $\Sigma = \{a, b\}$:

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

All regular languages over Σ can be constructed by combining these four languages in various ways by recursively applying the union, product and closure operations.

Example

Is the language $\{a, ab\}$ regular?
Yes, since it can be constructed from the four basic regular languages using a product and union operation–

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

We can also see from this that it is possible to construct any finite language in this way, and therefore **all finite languages are regular**. There are also many infinite languages which are regular.

Regular Expressions

If you wanted to find an algorithm which can determine if a string belongs to a particular regular language, you would most likely use a **regular expression**

> **Definition**
>
> A **Regular Expression** is basically a short-hand way of showing how a regular language is constructed from a base set of regular languages.
> For each regular expression E, there is a regular language $L(E)$

Like the regular languages discussed previously, regular expressions can be manipulated inductively to form new regular expressions.

- **Basis** - Λ, $\emptyset$ and a are regular expressions for all $a \in \Sigma$

- **Induction** - If R and E are also regular expressions, then (R), $R + E$, $R \cdot E$ and R^* are also regular expressions

> **Example**
>
> A few of the regular expressions over the alphabet $\Sigma = \{a, b\}$ are
>
> $$\Lambda, \emptyset, a, b$$
> $$\Lambda + b, b^*, a + (b \cdot a)$$
> $$(a + b) \cdot a, a \cdot b^*, a^* \cdot b^*$$

So that we don't have to include lots of parentheses, the operations have the following hierarchy–

- $*$ (Highest)

- $\cdot$

- $+$ (Lowest)

> **Example**
>
> The regular expression
> $$a + b \cdot a^*$$
> is the same as
> $$(a + (b \cdot (a^*)))$$

We can also juxtapose languages rather than using $\cdot$ where there is only one possible interpretation

> **Example**
>
> The regular expression
> $$a + b \cdot a^*$$
> is the same as
> $$a + ba^*$$

Operators

In regular expressions, there are two binary operations ($+$ and $\cdot$), and one unary operation ($*$). These are closely related to the union, product, and closure operations over corresponding languages.

> **Example**
>
> The regular expression
> $$a + bc^*$$
> is more-or-less shorthand for the regular language
> $$\{a\} \cup (\{b\} \cdot (\{c\}^*))$$

From Expressions to Languages

There is a very simple substitution method to find the language described by a regular expression.

> **Example**
>
> Find the language described by the regular expression $a + bc^* -$
>
> $$\begin{aligned}
> L(a + bc^*) &= L(a) \cup L(bc^*) \\
> &= L(a) \cup (L(b) \cdot L(c^*)) \\
> &= L(a) \cup (L(b) \cdot L(c)^*) \\
> &= \{a\} \cup (\{b\} \cdot \{c\}^*) \\
> &= \{a\} \cup (\{b\} \cdot \{\Lambda, c, c^2, c^3, \ldots\}) \\
> &= \{a\} \cup \{b, bc, bc^2, bc^3, \ldots\} \\
> &= \{a, b, bc, bc^2, bc^3, \ldots\}
> \end{aligned}$$

This approach can also be used to prove that a given language is regular.

Many infinite languages can easily be proven to be regular, by finding a regular expression that describes it. Not all infinite languages are regular however.

Regular expressions are also not necessarily unique, as they may represent the same language as another regular expression, such as $a + b$ and $b + a$ being different regular expressions, but both represent the language $\{a, b\}$

> **Definition**
>
> Regular expressions are said to be **Equal** if their languages are the same. If R and E are regular expressions where $L(R) = L(E)$, then they are equal and can be written as $R = E$.

Properties of Regular Expressions

There are many general equalities for regular expressions, which hold for any regular expressions R, E, and F, and can be proven by using the basic 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

$$R\emptyset = \emptyset R = \emptyset$$
$$R\Lambda = \Lambda R = R$$
$$(RE)F = R(EF)$$

- COME BACK HERE

Regular Grammars

There are several ways to characterize grammars which describe regular languages.

> **Definition**
>
> A **Regular Grammar** is one where all of the productions take one of the following forms
>
> $$B \rightarrow \Lambda$$
> $$B \rightarrow w$$
> $$B \rightarrow A$$
> $$B \rightarrow wA$$
>
> Where A and B are non-terminals and w is a non-empty string of terminals

Only one non-terminal can appear on the right hand side of any production. Non-terminals must appear on the right end of the right hand side. Therefore the productions $A \rightarrow aBc$ and $S \rightarrow TU$ are not productions in a regular grammar, but $A \rightarrow abcA$ is.

For any regular language, we can find a regular grammar which describes it, but there may also be non-regular grammars which produce it.

Lecture - A4: Finite Automata

13:00   10/10/24   Janka Chlebikova

There are several different 'models of computation', such as finite automata, push-down automata and turing machines. They are all abstract models of computers which are capable of different things. They all have an input tape with a single string of an infinite length and can accept or reject the input string.

Finite Automata

Finite automata are the most basic models of a computer.

Definition

A **Finite Automaton** has three components–
  - An input tape, which contains an input string over the alphabet Σ
  - A head, which reads the input string one symbol at a time
  - Memory, which is realised as a finite set Q of states, of which the automaton can only be in one at any time
The 'program' of the automaton defines how the read symbol changes the current state.

To better understand an automaton, they are typically represented as a transition graph, which is a form of directed graph.

Example
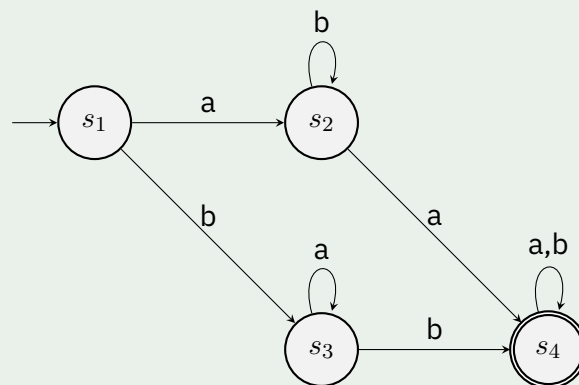
This is a finite automata which takes the alphabet $\Sigma = \{a, b\}$, and accepts strings such as $abba$, $baab$, etc.

Each automaton has one initial state, denoted by the arrow entering s_1, and at least one final or accepting state, denoted by the double circle such as s_4. Once the whole input string is read into the automaton, if the current state is a final or accepting state, then the string is accepted. Otherwise, the string is rejected. The language of an automaton is the set of strings it accepts.

A Formal Definition

> **Definition**
>
> A finite automaton is defined by
> - A set of states, Q
> - A start state, $s \in Q$
> - A set of accepting states, $Q_a \subset Q$
> - A set of transition functions $T : Q \times \Sigma \to Q$

> **Example**
>
> A very simple finite automaton could be defined by
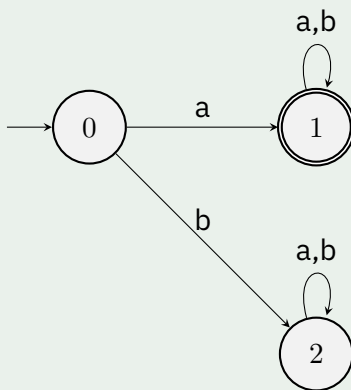>
> $$Q = \{0, 1, 2\}$$
> $$s = 0$$
> $$Q_a = \{1\}$$
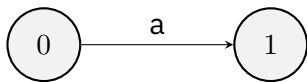> $$T(0, a) = 1, T(0, b) = 2$$
> $$T(1, a) = T(1, b) = 1$$
> $$T(2, a) = T(2, b) = 2$$
>
> And as a transition graph,

State Transition Functions

A state transition function, $T : Q \times \Sigma \to Q$, of the form

is represented by $T(0, a) = 1$, where $0, 1 \in Q$ and $a \in \Sigma$.

Deterministic Finite Automata

In the previous examples, there is always exactly one transition function for every state and symbol – every node has exactly one edge for each possible input symbol. Such automata are known as **deterministic** finite automata. That means that for any input string, we always know which unique state we are in at any symbol in the string.

> **Definition**
>
> A **DFA** accepts a string w over Σ^* if and only if there is a path from the initial state to an accepting state, such that w is the concatenation of the labels on the edges of the path. Otherwise, the DFA rejects w. For a FA to be deterministic, there must also be **exactly** one edge from each state for each symbol in Σ.
> The set of all strings over Σ accepted by the DFA M is known as the language of M, and is represented by $L(M)$.

It has been proven that "the class of regular languages is exactly the same as the class of languages accepted by DFAs". This means that for any given regular language, we can find at least one DFA which recognises it, and vice versa.

Non-Deterministic Finite Automata

With a DFA, you always know exactly which state it will be in after a given input string. This is not the case with an NFA, since in any given state there may be zero or more transitions for each symbol in the alphabet. Since more than one transition is possible for each state, it is impossible to know which state it will be in given an input string. If the NFA reaches an accepting state after reading the string, it is accepted. Otherwise it is rejected. Since there could be multiple paths for any given strings, as long as one or more path reaches an accepting state, it is accepted. Also, if there are no edges that can be traversed from a state using the current input symbol, the input string is immediately rejected.

Additionally, an edge may be labelled with Λ, which means that one can make the transition without consuming any symbols from the input string.

The non-determinism of these FA also means that they are impossible to run on a traditional computer, at least without using tricks that make them much less efficient. With the advent of quantum computers, it may become possible to use NFAs for much more efficient parsing and pattern matching.
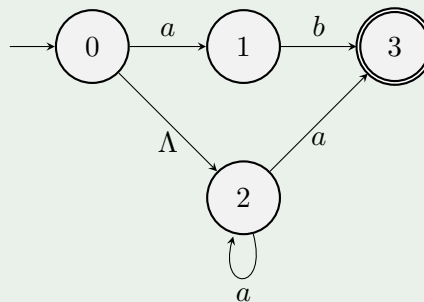
> **Example**
>
>
>
> This is an NFA which corresponds to the regular expression $ab + a^*a$.

Since we are able to have multiple edges with the same symbol from a node, we could also use this different NFA which recognises the language of the same regular expression.
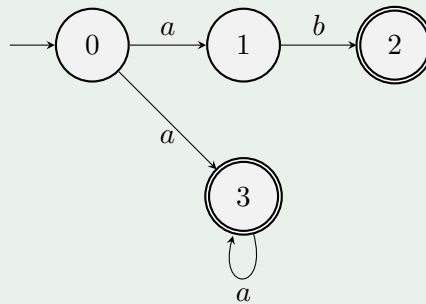
> **Example**
>
>

NFA Transition Functions

Since NFAs are non-deterministic, their transition functions instead take the form $T : Q \times \Sigma \to P(Q)$. This allows for the result of any given transition function to be a set of states which could be transitioned to.

> **Example**
>
> Taking the first example of an NFA, the transition function might look as below–
>
> $$T : Q \times \Sigma \to P(Q)$$
> $$T(0, a) = \{1\}, T(0, \Lambda) = \{2\}$$
> $$T(1, b) = \{3\}$$
> $$T(2, a) = \{2, 3\}$$
>
> and for the second example–
>
> $$T : Q \times \Sigma \to P(Q)$$
> $$T(0, a) = \{1, 3\}$$
> $$T(1, b) = \{2\}$$
> $$T(3, a) = \{3\}$$

DFAs vs NFAs

On the surface it may seem like there aren't many differences between NFAs and DFAs, and in fact, DFAs are a subset of NFAs. It has also been proven that "the class of languages accepted by NFAs is also exactly the class of regular languages", just as for DFAs. This also means that for every NFA there must be an equivalent DFA, or at least one that accepts the same language.

Generally, NFAs are easier to construct for any given regular expression, and tend to be more simple, with fewer states and transitions. However, DFAs are much easier to execute, especially on the deterministic computers we currently use. Since they recognise the same set of languages, it is always possible to find equivalents, though the DFA will usually be much larger and more complex.

Finding an Equivalent DFA

Generally, the DFA will have more states than the equivalent NFA. If the DFA has n states, the DFA may have as many as 2^n states. To actually do this conversion, you can follow the algorithm below (shown here converting the NFA in the figure below)
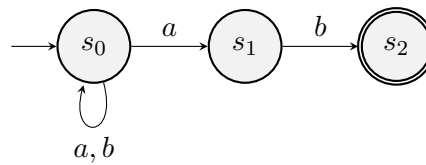
Figure 5.1: The NFA which accepts the language for the regular expression $(a + b)^* ab$

Step 1 – Begin with the NFA start state. If it's connected to any other states by Λ, make the initial state of the DFA a set containing the NFA's initial state and those others.
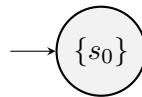

Figure 5.2: The resulting DFA after step 1

Step 2 – For each symbol, determine the set of possible NFA states you may be in after reading it. This set becomes the label of the state, and is connected to the previous state by that symbol. Only create a new state if one with that label does not already exist.
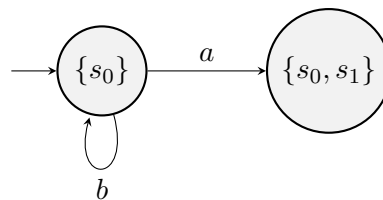

Figure 5.3: The resulting DFA after step 2

Step 3 – Repeat Step 2 for each new DFA state, until the system is closed. DFA accepting states are those that include one or more NFA accepting states. If there is no transition for a symbol in the NFA, create a new state in the DFA with the label $\emptyset$, and add loops for all symbols (known as a non-final trap state).
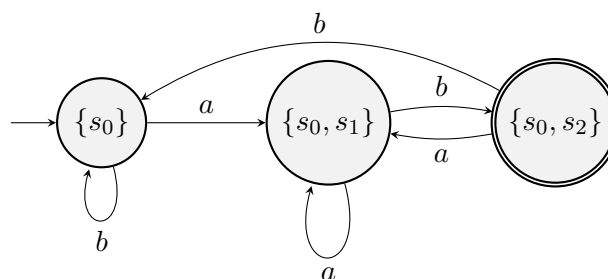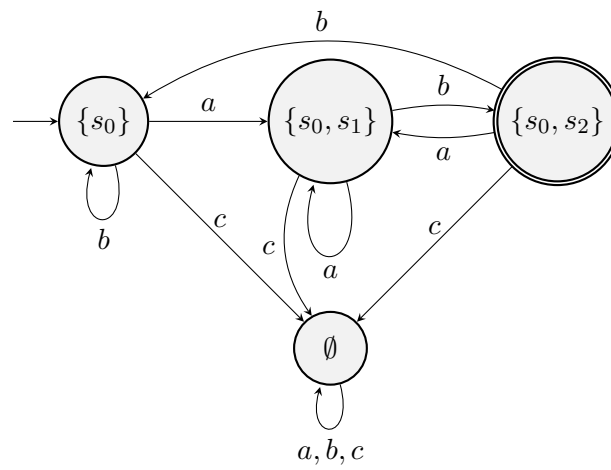

Figure 5.4: The final DFA

Figure 5.5: The same DFA, assuming that the language was $\{a, b, c\}$ rather than $\{a, b\}$