Formal Languages and Compilers

Jordan Pyott

2022-01-21

Contents

Course Information

Course Staff

- Coordinator/Lecturer
  - Walter Guttmann
    - 033692451
    - walter.guttmann@canterbury.ac.nz

Assessments and Grading

Grading policy

1. You mist achieve an average grade of at least 50% over all assessment items
2. You mist achieve an average grade of at least 45% over all invigilated assessments

Assessment Items

- Quizzes (15%)
- Assignment Superquiz (10%)
- Lab Test (15%)
- Final Exam (60%)

Textbooks/Resources

No textbooks are required, but see the following book for additional information:

- Carol Critchlow and David Eck; Foundationals of Computation; version 2.3.1, 2011

Lectures

Lecture One - Introduction

Topic overview of the course

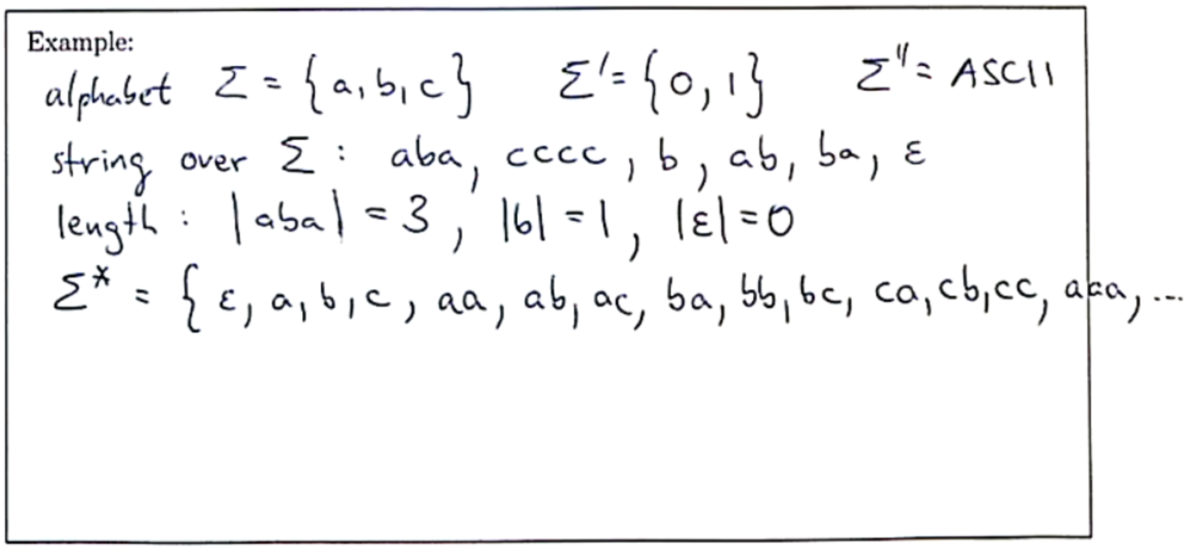
- pattern matching
    - Regular expressions describe patterns
    - Search using REGEX is supported in many programs
    - Can all patterns be described by regular expressions?
    - One to one with state diagrams and automata
- Compilers
    - Programs can be run by an interpreter or by compiling them first
    - Interpreting may be slow
    - Compiling to machine code avoids much of the overhead
    - Compiler performs analysis, code generation and optimisation
    - How can these tasks be automated for different programming languages?
- Syntax analysis
    - Analyses code to determine if the syntax is correct for compiling, this is done by using context free grammers *there are other methods of doing this however this is the main one we will look at in this course*
    - *does the syntax conform to the languages grammar?*
    - Ideally we want to generate the parser for our language, we will look into how to manually like are parser and how regular expressions and pattern matching can be used to evaluate this behaviour.
- Code generation
    - There are formalisms that exist to generate code in order to create compilers via code generation.

Lecture Two - Finite Automata and Regular Languages: Symbols, Strings and Languages

Languages

- An alphabet Σ is non-empty finite set of *symbols/*
- A string over Σ is a finite sequence of symbols from Σ

- The length of $|\sigma|$ of a string σ is the number of symbols in σ
- The empty string ϵ is the unique set of length 0
- $\Sigma^\cdot$ is the set of all strings over Σ
- A language L over Σ is a set of strings $L \subseteq \Sigma^\cdot$



Example:

alphabet $\Sigma = \{a, b, c\}$ $\Sigma' = \{0, 1\}$ $\Sigma'' =$ ASCII

string over Σ : aba, cccc, b, ab, ba, ε

length : $|aba| = 3$, $|b| = 1$, $|\varepsilon| = 0$

$\Sigma^* = \{ \varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aba, \ldots$

Figure 1: Example

- Note that with a finite alphabet we can have an infinite size for $\Sigma^\cdot$
  - This is because we have not specified a size for our length of elements within $\Sigma^\cdot$

> An example of a set that we might use is the unicode set as Sigma.

For example:

- $python \subset UNICODE$
- $english \subset UNICODE$

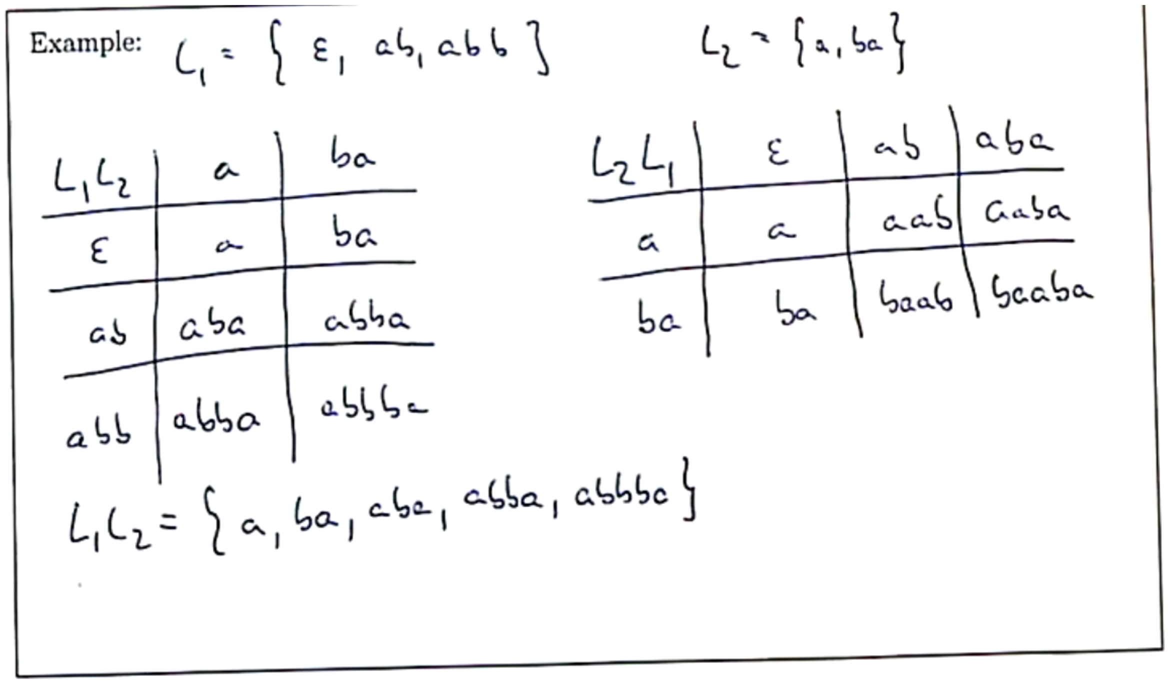
Because of this relationship, we can use filtering, searching and REGEX in order to manipulate and set rules around this relationship (or syntax in the case of programming languages by using comparisons and combination of formalisms.

Let $a, b \in \Sigma$ be symbols and let $x, y, z \in \Sigma$ be strings. - Symbols and strings can be concatenated by writing one after the other - xy is the concatenated version of x and y. - Note that concatenation is accociative - ϵ is an identity for concatenation $\epsilon x = x = x\epsilon$ - $|xy| = |x| + |y|$

Lifting to a set

$Let A, B \subseteq \Sigma^{cdot}$ be languages:

- concatenate languages A and B by conciliating each string from A with each from B
- $AB = \{xy|x \in A, y \in B\}$
- Language concatenation is associative
- $\{\epsilon\}$ is the identity of language concatenation



Example: $L_1 = \{\varepsilon, ab, abb\}$ $L_2 = \{a, ba\}$

L_1L_2	a	ba
ε	a	ba
ab	aba	abba
abb	abba	abbba

L_2L_1	ε	ab	aba
a	a	aab	aaba
ba	ba	baab	baaba

$L_1L_2 = \{a, ba, aba, abba, abbba\}$

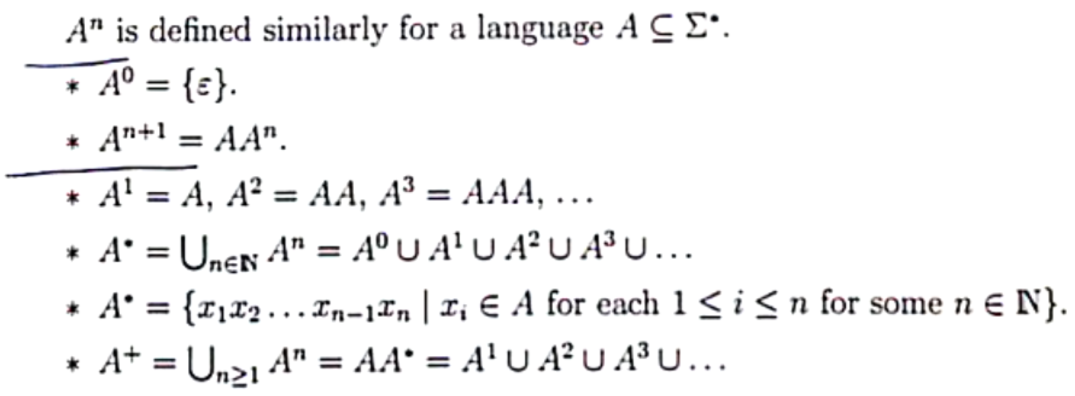
Figure 2: Lifting Example

Concatenation can be iterated

- a^n is the string comprising n copies of the symbol $a \in \Sigma$
- x^n is the string that concatinates n copies of the string $x \in \Sigma$
- These operations are defined *inductively*
- The base case is $x^0 = \epsilon$
- The *inductive case* is $x^{n+1} = x^n x$

Example: $a^5 = aaaaa$

> Take aways, the * symbol means that we have zero or more of something, + means that we have one or more of something (this is how we use this notation practically in regex expressions



A^n is defined similarly for a language $A \subseteq \Sigma^*$.

* $A^0 = \{\varepsilon\}$.
* $A^{n+1} = AA^n$.
* $A^1 = A$, $A^2 = AA$, $A^3 = AAA$, $\ldots$
* $A^* = \bigcup_{n \in \mathbb{N}} A^n = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \ldots$
* $A^* = \{x_1 x_2 \ldots x_{n-1} x_n \mid x_i \in A \text{ for each } 1 \leq i \leq n \text{ for some } n \in \mathbb{N}\}$.
* $A^+ = \bigcup_{n \geq 1} A^n = AA^* = A^1 \cup A^2 \cup A^3 \cup \ldots$

Figure 3: Powers of Language