Lecture Notes For

Formal Languages and Automata

Gordon J. Pace

December 1997

Department of Computer Science & A.I.
Faculty of Science
University of Malta

Draft

Contents

Chapter 1

Introduction and Motivation

1.1 Introduction

What is a language? Whether we restrict our discussion to natural languages such as English or Maltese, or whether we also discuss artificial languages such as computer languages and mathematics, the answer to the question can be split into two parts:

Syntax: An English sentence of the form:

$$\langle \textit{noun-phrase} \rangle \ \langle \textit{verb} \rangle \ \langle \textit{noun-phrase} \rangle$$

(such as *The cat eats the cheese*) has a correct structure (assuming that the verb is correctly conjugated). On the other hand, a sentence of the form $\langle \textit{adjective} \rangle$ $\langle \textit{verb} \rangle$ (such as *red eat*) does not make sense structurally. A sentence is said to be syntactically correct if it is built from a number of components which make structural sense.

Semantics: Syntax says nothing about the meaning of a sentence. In fact a number of syntactically correct sentences make no sense at all. The classic example is a sentence from Chomsky:

Colourless green ideas sleep furiously

Thus, at a deeper level, sentences can be analyzed semantically to check whether they make any sense at all.

In mathematics, for example, all expressions of the form x/y are usually considered to be syntactically correct, even though $10/0$ usually corresponds to no meaningful semantic interpretation.

In spoken natural language, we regularly use syntactically incorrect sentences, even though the listener usually manages to 'fix' the syntax and understand the underlying semantics as meant by the speaker. This is particularly evident in baby-talk. At least babies can be excused, however poets have managed to make an art out of it!

There was a poet from Zejtun
Whose poems were strange, out-of-tune,
Correct metric they had,
The rhyme wasn't that bad,
But his grammar sometimes like baboon.

For a more accomplished poet and author, may I suggest you read one of the poems in Lewis Carroll's 'Alice Through the Looking Glass', part of which goes:

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

In computer languages, a compiler generates errors of a syntactic nature. Semantic errors appear at run-time when the computer is interpreting the compiled code.

In this course we will be dealing with the syntactic correctness of languages. Semantics shall be dealt with in a different course.

However, splitting the problem into two, and choosing only one thread to follow, has not made the solution much easier. Let us start by listing a number of properties which seem evident:

- a number of words (or symbols or whatever) are used as the basic building blocks of a language. Thus in mathematics, we may have numbers whereas in English we would have English words.

- certain sequences of the basic symbols are allowed (are valid sentences in the language) whereas others are not.

This characterizes completely what a language is: a set of sequences of symbols. So for example, English would be the set:

$$\{\text{The cat ate the mouse}, \text{I eat}, \ldots\}$$

Note that this set is infinite (*The house next to mine is red*, *The house next to the one next to mine is red*, *The house next to the one next to the one next to mine is red*, etc are all syntactically valid English sentences), however, it does not include certain sequences of words (such as *But his grammar sometimes like baboon*).

Enlisting an infinite set is not the most pleasant of jobs. Furthermore, this also creates a paradox. Our brains are of finite size, so how can they contain an infinite language? The solution is actually rather simple: the languages we are mainly interested in can be generated from a finite number of rules. Thus, we need not remember whether *I eat* is syntactically valid, but we mentally apply a number of rules to *deduce* its validity.

Languages with which we are concerned are thus a finite set of basic symbols, together with a finite set of rules. These rules, the grammar of the language, allow us to generate sequences of the basic symbols (usually called the alphabet). Sequences we can generate are syntactically correct sentences of the languages, whereas ones we cannot are syntactically incorrect.

Valid Pascal variable names can be generated from the letters and numbers ('`A`' to '`Z`', '`a`' to '`z`' and '`0`' to '`9`') and the underscore symbol ('_'). A valid variable name starts with a letter and may then follow with any number of symbols from the alphabet. Thus, `I_am_not_a_variable` is a valid variable name, whereas `25_lettered_variable_name` is not.

The next problem to deal with is the representation of language's grammar. Various solutions have been proposed and tried out. Some are simply incapable of describing certain desirable grammars. Of the more general representations some are better suited than others to be used in certain circumstances.

1.2 Grammar Representation

Computer language manuals usually use the BNF (Backus-Naur Form) notation to describe the syntax of the language. The grammar of the variable names as described earlier can be given as:

$$
\begin{array}{lcl}
\langle \textit{letter} \rangle & ::= & \texttt{a} \mid \texttt{b} \mid \ldots \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \ldots \texttt{Z} \\
\langle \textit{number} \rangle & ::= & \texttt{0} \mid \texttt{1} \mid \ldots \texttt{9} \\
\langle \textit{underscore} \rangle & ::= & \texttt{\_} \\
\langle \textit{misc} \rangle & ::= & \langle \textit{letter} \rangle \mid \langle \textit{number} \rangle \mid \langle \textit{underscore} \rangle \\
\langle \textit{end-of-var} \rangle & ::= & \langle \textit{misc} \rangle \mid \langle \textit{misc} \rangle \langle \textit{end-of-var} \rangle \\
\langle \textit{variable} \rangle & ::= & \langle \textit{letter} \rangle \mid \langle \textit{letter} \rangle \langle \textit{end-of-var} \rangle
\end{array}
$$

The rules are read as definitions. The '::=' symbol is the definition symbol, the | symbol is read as 'or' and adjacent symbols denote concatenation. Thus, for example, the last definition states that a string is a valid ⟨*variable*⟩ if it is either a valid ⟨*letter*⟩ or a valid ⟨*letter*⟩ concatenated with a valid ⟨*end-of-var*⟩. The names in angle brackets (such as ⟨*letter*⟩) do not appear in valid strings but may be used in the derivation of such strings. For this reason, these are called non-terminal symbols (as opposed to terminal symbols such as `a` and `4`).

Another representation frequently used in computing especially when describing the options available when executing a program looks like:

`cp [-r] [-i]` ⟨*file-name*⟩ ⟨*file-name*⟩|⟨*dir-name*⟩

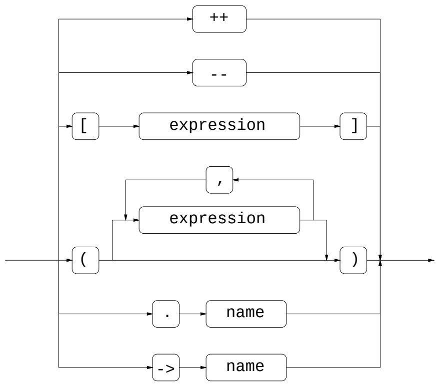
This partially describes the syntax of the copy command in UNIX. It says that a copy command starts with the string `cp`. It may then be followed by `-r` and similarly `-i` (strings enclosed in square brackets are optional). A filename is then given followed by a file or directory name (| is choice).

Sometimes, this representation is extended to be able to represent more useful languages. A string followed by an asterisk is used to denote 0 or more repetitions of that string, and a string followed by a plus sign used to denote 1 or more repetitions. Bracketing may also be used to make precedence clear. Thus, valid variable names may be expressed by:

```
a|b ...Z (a|b ...Z|_|0|1| ...9)*
```

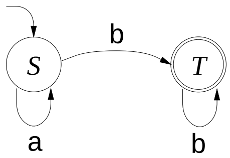
Needless to say, it quickly becomes very complicated to read expressions written in this format.

Another standard way of expressing programming language portions is by using syntax diagrams (sometimes called 'railroad track' diagrams). Below is a diagram taken from a book on standard C to define a postfix operator.



Even though you probably do not have the slightest clue as to what a postfix operator is, you can deduce from the diagram that it is either simply ++, or −−, or an expression within square brackets, or a list of expressions in brackets or a name preceded by a dot or by `->`. The definition of expressions and names would be similarly expressed. The main strength of such a system is ease of comprehension of a given grammar.

Another graphical representation uses finite state automata. A finite state automaton has a number of different named states drawn as circles. Labeled arrows are also drawn starting from and ending in states (possibly the same). One of the states is marked as the starting state (where computation starts), whereas a number of states are marked as final states (where computation may end). The initial state is marked by an incoming arrow and the final states are drawn using two concentric circles. Below is a simple example of such an automaton:



The automaton starts off from the initial state and, upon receiving an input, moves along an arrow originating from the current state whose label is the same as the given input. If no such arrow is available, the machine may be seen to 'break'. Accepted strings are ones for which, when given as input to the machine, result in a computation starting from the initial state and ending in a terminal state without breaking the machine. Thus, in our example above `b` is accepted, as is `ab` and `aabb`. If we denote n repetitions of a string $\mathtt{s}$ by $\mathtt{s}^n$, we notice that the set of strings accepted by the automaton is effectively:

$$\{\mathtt{a}^n\mathtt{b}^m \mid n \geq 0 \wedge m \geq 1\}$$

1.3 Discussion

These different notations give rise to a number of interesting issues which we will discuss in the course of the coming lectures. Some of these issues are:

- How can we formalize these language definitions? In other words, we will interpret these definitions mathematically in order to allow us to reason formally about them.
- Are some of these formalisms more expressive than others? Are there languages expressible in one but not another of these formalisms?
- Clearly, some of the definitions are simpler to implement as a computer program than others. Can we define a translation of a grammar from one formalism to another, thus enabling us to implement grammars expressed in a difficult-to-implement notation by first translating them into an alternative formalism?
- Clearly, even within the same formalism, certain languages can be expressed in a variety of ways. Can we define a simplifying procedure which simplifies a grammar (possibly as an aid to implementation?)
- Again, given that some languages can be expressed in different ways in the same formalism, is there some routine way by which we can compare two grammars and deduce their (in)equality?

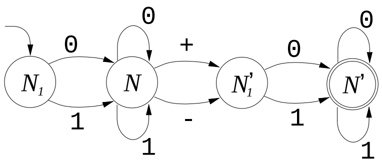
On a more practical level, at the end of the course you should have a better insight into compiler writing. At least, you will be familiar with the syntax checking part of a compiler. You should also understand the inner workings of LEX and YACC (standard compiler writing tools).

1.4 Exercises

1. What strings of type $\langle S \rangle$ does the following BNF specification accept?

$$\begin{array}{lll} \langle A \rangle & ::= & \texttt{a}\langle B \rangle \mid \texttt{a} \\ \langle B \rangle & ::= & \texttt{b}\langle A \rangle \mid \texttt{b} \\ \langle S \rangle & ::= & \langle A \rangle \mid \langle B \rangle \end{array}$$

2. What strings are accepted by the following finite state automaton?



3. A palindrome is a string which reads front-to-back the same as back-to-front. For example `anna` is a palindrome, as is `madam`. Write a BNF notation which accepts exactly all those palindromes over the characters `a` and `b`.

4. The correct (restricted) syntax for `write` in Pascal is as follows:

   The instruction `write` is always followed by a parameter-list enclosed within brackets. A parameter-list is a comma-separated list of parameters, where each parameter is either a string in quotes or a variable name.

   (a) Write a BNF specification of the syntax

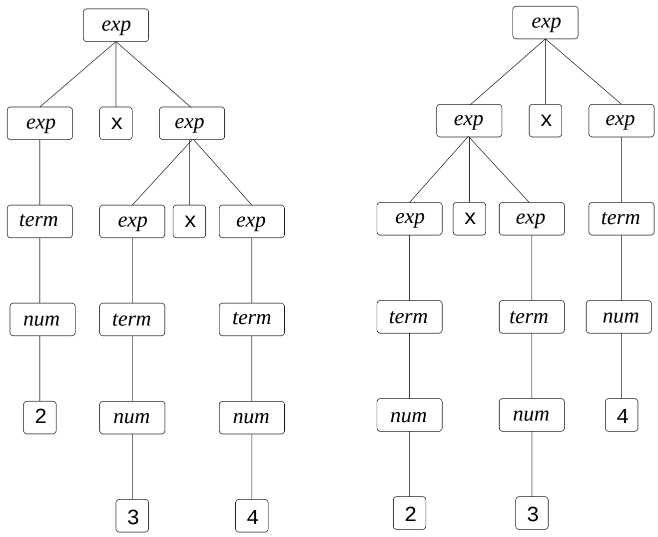
   (b) Draw a syntax diagram

   (c) Draw a finite state automaton which accept correct `write` statements (and nothing else)

5. Consider the following BNF specification portion:

$$\begin{array}{lll} \langle \mathit{exp} \rangle & ::= & \langle \mathit{term} \rangle \mid \langle \mathit{exp} \rangle \times \langle \mathit{exp} \rangle \mid \langle \mathit{exp} \rangle \div \langle \mathit{exp} \rangle \\ \langle \mathit{term} \rangle & ::= & \langle \mathit{num} \rangle \mid \ldots \end{array}$$

   An expression such as $2 \times 3 \times 4$ can be accepted in different ways. This becomes clear if we draw a tree to show how the expression has been parsed. The two different trees for $2 \times 3 \times 4$ are given below:



Clearly, different acceptance routes may have different meanings. For example $(1 \div 2) \div 2 = 0.25 \neq 1 = 1 \div (2 \div 2)$. Even though we are currently oblivious to issues regarding the semantics of a language, we identify grammars in which there are sentences which can be accepted in alternative ways. These are called ambiguous grammars.

In natural language, these ambiguities give rise to amusing misinterpretations.

Today we will discuss sex with Margaret Thatcher

In computer languages, however, the results may not be as amusing. Show that the following BNF grammar is ambiguous by giving an example with the relevant parse trees:

$$
\begin{array}{lcl}
\langle \mathit{program} \rangle & ::= & \texttt{if } \langle \mathit{bool} \rangle \texttt{ then } \langle \mathit{program} \rangle \\
& | & \texttt{if } \langle \mathit{bool} \rangle \texttt{ then } \langle \mathit{program} \rangle \texttt{ else } \langle \mathit{program} \rangle \\
& | & \vdots
\end{array}
$$

Chapter 2

Languages and Grammars

2.1 Introduction

Recall that a language is a (possibly infinite) set of strings. A grammar to construct a language can be defined in terms of two pieces of information:

- A finite set of symbols which are used as building blocks in the construction of valid strings, and
- A finite set of rules which can be used to deduce strings. Strings of symbols which can be derived from the rules are considered to be strings in the language being defined.

The aim of this chapter is to formalize the notions of a language and a grammar. By defining these concepts mathematically we then have the tools to prove properties pertaining to these languages.

2.2 Alphabets and Strings

Definition: An *alphabet* is a finite set of symbols. We normally use variable Σ for an alphabet. Individual symbols in the alphabet will normally be represented by variables a and b.

Note that with each definition, I will be including what I will normally use as a variable for the defined term. Consistent use of these variable names should make proofs easier to read.

Definition: A *string* over an alphabet Σ is simply a finite list of symbols from Σ. Variables normally used are s, t, x and y.

The set of all strings over an alphabet Σ is usually written as Σ^*.

To make the expression of strings easier, we write their components without separating commas or surrounding brackets, Thus, for example, $[h, e, l, l, o]$ is usually written as $hello$.

What about the empty list? Since the empty string simply disappears when using this notation, we use symbol ε to represent it.

Definition: Juxtaposition of two strings is the concatenation of the two strings. Thus:

$$st \stackrel{def}{=} s +\!\!+\, t$$

This notation simplifies considerably the presentation of strings: Concatenating $hello$ with $world$ is written as $helloworld$, which is precisely the result of the concatenation!

Definition: A string s raised to a numeric power n (s^n) is simply the catenation of n copies of s. This can be defined by:

$$\begin{array}{rcl} s^0 & \stackrel{def}{=} & \varepsilon \\ s^{n+1} & \stackrel{def}{=} & ss^n \end{array}$$

Definition: The length of a string s, written as $|s|$, is defined by:

$$\begin{array}{rcl} |\varepsilon| & \stackrel{def}{=} & 0 \\ |ax| & \stackrel{def}{=} & 1 + |x| \end{array}$$

Note that $a \in \Sigma$ and $x \in \Sigma^*$.

Definition: String s is said to be a *prefix* of string t, if there is some string w such that $t = sw$.

Similarly, string s is said to be a *postfix* of string t, if there is some string w such that $t = ws$.

2.3 Languages

Definition: A *language* defined over an alphabet Σ is simply a set of strings over the alphabet. We normally use variable L to stand for a language.

Thus, L is a language over Σ if and only if $L \subseteq \Sigma^*$.

Definition: The catenation of two languages L_1 and L_2, written as L_1L_2 is simply the set of all strings which can be split into two parts: the first being in L_1 and the second in L_2.

$$L_1L_2 \stackrel{def}{=} \{st \mid s \in L_1 \wedge t \in L_2\}$$

Definition: As with strings, we can define the meaning of raising a language to a numeric power:

$$\begin{aligned} L^0 &\stackrel{def}{=} \{\varepsilon\} \\ L^{n+1} &\stackrel{def}{=} LL^n \end{aligned}$$

Definition: The Kleene closure of a language, written as L^* is simply the set of all strings which are in L^n, for some values of n:

$$L^* \stackrel{def}{=} \bigcup_{n=0}^{\infty} L^n$$

L^+ is the same except that n must be at least 1:

$$L^+ \stackrel{def}{=} \bigcup_{n=1}^{\infty} L^n$$

Some laws which these operations enjoy are listed below:

$$\begin{aligned} L^+ &= LL^* \\ L^+ &= L^*L \\ L^+ \cup \{\varepsilon\} &= L^* \\ (L_1 \cup L_2)L_3 &= L_1L_3 \cup L_2L_3 \\ L_1(L_2 \cup L_3) &= L_1L_2 \cup L_1L_3 \end{aligned}$$

The proof of these equalities follows the standard way of checking equality of sets: To prove that $A = B$, we prove that $A \subseteq B$ and $B \subseteq A$.

Example: Proof of $L^+ = LL^*$

$$
\begin{array}{ll}
 & x \in LL^* \\
\Leftrightarrow & \text{definition of } L^* \\
 & x \in L \bigcup\limits_{n=0}^{\infty} L^n \\
\Leftrightarrow & \text{definition of concatenation} \\
 & x = x_1x_2 \wedge x_1 \in L \wedge x_2 \in \bigcup\limits_{n=0}^{\infty} L^n \\
\Leftrightarrow & \text{definition of union} \\
 & x = x_1x_2 \wedge x_1 \in L \wedge \exists m \geq 0 \cdot x_2 \in L^m \\
\Leftrightarrow & \text{predicate calculus} \\
 & \exists m \geq 0 \cdot x = x_1x_2 \wedge x_1 \in L \wedge x_2 \in L^m \\
\Leftrightarrow & \text{definition of concatenation} \\
 & \exists m \geq 0 \cdot x \in LL^m \\
\Leftrightarrow & \text{definition of } L^{m+1} \\
 & \exists m \geq 0 \cdot x \in L^{m+1} \\
\Leftrightarrow & \exists m \geq 1 \cdot x \in L^m \\
\Leftrightarrow & \text{definition of union} \\
 & x \in \bigcup\limits_{n=1}^{\infty} L^n \\
\Leftrightarrow & \text{definition of } L^+ \\
 & x \in L^+
\end{array}
$$

2.3.1 Exercises

1. What strings do the following languages include:
   (a) $\{a\}\{aa\}^*$
   (b) $\{aa,\ bb\}^* \cap (\{a\}^* \cup \{b\}^*)$
   (c) $\{a,\ b,\ \ldots z\}\{a,\ b,\ \ldots z,\ 0,\ \ldots 9\}^*$
2. What are $L\emptyset$ and $L\{\varepsilon\}$?

3. Prove the four unproven laws of language operators.
4. Show that the laws about catenation and union do not apply to catenation and intersection by finding a counter-example which shows that $(L_1 \cap L_2)L_3 \neq L_1L_3 \cap L_2L_3$.

2.4 Grammars

A grammar is a finite mechanism which we will use to generate potentially infinite languages.

The approach we will use is very similar to BNF. The strings we generate will be built from symbols in a particular alphabet. These symbols are sometimes referred to as terminal symbols. A number of *non-terminal* symbols will be used in the computation of a valid string. These appear in our BNF grammars within angle brackets. Thus, for example, in the following BNF grammar, the alphabet is $\{a,\ b\}$[1] and the non-terminal symbols are $\{\langle W\rangle,\ \langle A\rangle,\ \langle B\rangle\}$.

$$
\begin{array}{lcl}
\langle W\rangle & ::= & \langle A\rangle \mid \langle B\rangle \\
\langle A\rangle & ::= & \mathrm{a}\langle A\rangle \mid \varepsilon \\
\langle B\rangle & ::= & \mathrm{b}\langle B\rangle \mid \varepsilon
\end{array}
$$

The BNF grammar is defining a number of transition rules from a non-terminal symbol to strings of terminal and non-terminal symbols. We choose a more general approach, where transition rules transform a non-empty string into another (potentially empty) string. These will be written in the form: $from \rightarrow to$. Thus, the above BNF grammar would be represented by the following set of transitions:

$$\{W \rightarrow A|B,\ A \rightarrow aA|\varepsilon,\ B \rightarrow bB|\varepsilon\}$$

Note that any rule of the form $\alpha \rightarrow \beta|\gamma$ can be transformed into two rules of the form $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$.

Only one thing remains. If we were to be given the BNF grammar just presented, we would be unsure as to whether we are to accept strings which can be derived from $\langle A\rangle$ or from $\langle B\rangle$ or from $\langle W\rangle$. It is thus necessary to specify which non-terminal symbols derivations are to start from.

[1] Actually any set which includes both a and b but not any non-terminal symbols

Definition: A *phrase structure grammar* is a 4-tuple $\langle \Sigma,\ N,\ S,\ P\rangle$ where:

Σ is the alphabet over which the grammar generates strings.
N is a set of non-terminal symbols.
S is one particular non-terminal symbol.
P is a relation of type $(\Sigma \cup N)^+ \times (\Sigma \cup N)^*$.

It is assumed that $\Sigma \cap N = \emptyset$.

Variables for non-terminals will be represented by uppercase letters and a mixture of terminal and non-terminal symbols will usually be represented by greek letters. G is usually used as a variable ranging over grammars.

The BNF grammar already given can thus be formalized to the phrase structure grammar $G = \langle \Sigma,\ N,\ S,\ P\rangle$, where:

$$\begin{array}{rcl}
\Sigma & = & \{a,\ b\} \\
N & = & \{W,\ A,\ B\} \\
S & = & W \\
P & = & \{\ \ W \to A, \\
 & & \phantom{\{\ \ } W \to B, \\
 & & \phantom{\{\ \ } A \to aA, \\
 & & \phantom{\{\ \ } A \to \varepsilon, \\
 & & \phantom{\{\ \ } B \to bB, \\
 & & \phantom{\{\ \ } B \to \varepsilon \quad \}
\end{array}$$

We still have to formalize what we mean by a particular string being generated by a certain grammar.

Definition: A string β is said to derive immediately from a string α in grammar G, written $\alpha \Rightarrow_G \beta$, if we can apply a production rule of G on a substring of α obtaining β. Formally:

$$\begin{array}{rcl}
\alpha \Rightarrow_G \beta & \stackrel{def}{=} & \exists \alpha_1, \alpha_2, \alpha_3, \gamma \cdot \\
 & & \quad \alpha = \alpha_1\alpha_2\alpha_3 \quad \wedge \\
 & & \quad \beta = \alpha_1\gamma\alpha_3 \quad \wedge \\
 & & \quad \alpha_2 \to \gamma \in P
\end{array}$$

Thus, for example, in the grammar we obtained from the BNF specification, we can prove that:

$$\begin{aligned} S &\Rightarrow_G aA \\ S &\Rightarrow_G bB \\ aaaA &\Rightarrow_G aaa \\ aaaA &\Rightarrow_G aaaaA \\ SAB &\Rightarrow_G SB \end{aligned}$$

But not that:

$$\begin{aligned} S &\Rightarrow_G a \\ A &\Rightarrow_G a \\ SAB &\Rightarrow_G aAAB \\ B &\Rightarrow_G B \end{aligned}$$

In particular, even though $A \Rightarrow_G aA \Rightarrow_G a$, it is not the case that $A \Rightarrow_G a$. With this in mind, we define the following relational closures of $\Rightarrow_G$:

$$\begin{aligned} \alpha \overset{0}{\Rightarrow}_G \beta &\overset{def}{=} \alpha = \beta \\ \alpha \overset{n+1}{\Rightarrow}_G \beta &\overset{def}{=} \exists \gamma \cdot \alpha \Rightarrow_G \gamma \wedge \gamma \overset{n}{\Rightarrow}_G \beta \\ \alpha \overset{*}{\Rightarrow}_G \beta &\overset{def}{=} \exists n \geq 0 \cdot \alpha \overset{n}{\Rightarrow}_G \beta \\ \alpha \overset{+}{\Rightarrow}_G \beta &\overset{def}{=} \exists n \geq 1 \cdot \alpha \overset{n}{\Rightarrow}_G \beta \end{aligned}$$

It can thus be proved that:

$$\begin{aligned} S &\overset{*}{\Rightarrow}_G a \\ A &\overset{*}{\Rightarrow}_G a \\ SAB &\overset{*}{\Rightarrow}_G aAAB \\ B &\overset{*}{\Rightarrow}_G B \end{aligned}$$

although it is not the case that $B \stackrel{+}{\Rightarrow}_G B$.

Definition: A string $\alpha \in (N \cup \Sigma)^*$ is said to be in sentential form in grammar G if it can be derived from S, the start symbol of G. $\mathcal{S}(G)$ is the set of all sentential forms in G:

$$\mathcal{S}(G) \stackrel{def}{=} \{\alpha : (N \cup \Sigma)^* \mid S \stackrel{*}{\Rightarrow}_G \alpha\}$$

Definition: Strings in sentential form built solely from terminal symbols are called *sentences*.

These definitions indicate clearly what we mean by the language generated by a grammar G. It is simply the set of all strings of terminal symbols which can be derived from the start symbol in any number of steps.

Definition: The language generated by grammar G, written as $\mathcal{L}(G)$ is the set of all sentences in G:

$$\mathcal{L}(G) \stackrel{def}{=} \{x : \Sigma^* \mid S \stackrel{+}{\Rightarrow}_G x\}$$

Proposition: $\mathcal{L}(G) = \mathcal{S}(G) \cap \Sigma^*$

For example, in the BNF example, we should now be able to prove that the language described by the grammar is the set of all strings of the form a^n and b^n, where $n \geq 0$.

$$\mathcal{L}(G) = \{a^n \mid n \geq 0\} \cup \{b^n \mid n \geq 0\}$$

Now consider the alternative grammar $G' = \langle \Sigma',\ N',\ S',\ P' \rangle$:

$$\begin{array}{rcl}
\Sigma' & = & \{a,\ b\} \\
N' & = & \{W,\ A,\ B\} \\
S' & = & W \\
P' & = & \{\ \ W \rightarrow A, \\
 & & \phantom{\{}\ \ W \rightarrow B, \\
 & & \phantom{\{}\ \ W \rightarrow \varepsilon, \\
 & & \phantom{\{}\ \ A \rightarrow a, \\
 & & \phantom{\{}\ \ A \rightarrow Aa, \\
 & & \phantom{\{}\ \ aAa \rightarrow a, \\
 & & \phantom{\{}\ \ B \rightarrow bB, \\
 & & \phantom{\{}\ \ B \rightarrow b \qquad \}
\end{array}$$

With some thought, it should be obvious that $\mathcal{L}(G) = \mathcal{L}(G')$. This gives rise to a convenient way of comparing grammars — by comparing the languages they produce.

Definition: The grammars G_1 and G_2 are said to be *equivalent* if they produce the same language: $\mathcal{L}(G_1) = \mathcal{L}(G_2)$.

2.4.1 Exercises

1. Show how aaa can be derived in G.

2. Show two alternative ways in which aaa can be derived in G'.

3. Give a grammar which produces only (and all) palindromes over the symbols $\{a,\ b\}$.

4. Consider the alphabet $\Sigma = \{+,\ =,\ \cdot\}$. Repetitions of $\cdot$ are used to represent numbers ($\cdot^n$ corresponding to n). Define a grammar to produce all valid sums such as $\cdot\cdot+\cdot=\cdot\cdot\cdot$.

5. Define a grammar which accepts strings of the form $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$ (and no other strings).

2.5 Properties and Proofs

The main reason behind formalizing the concept of languages and grammars within a mathematical framework is to allow formal reasoning about these entities.

A number of different techniques are used to prove different properties. However, basically all proofs use induction in some way or another.

The following examples attempt to show different techniques as used in proofs of different properties or grammars. It is however, very important that other examples are tried out to experience the 'discovery' of a proof, which these examples cannot hope to convey.

2.5.1 A Simple Example

We start off with a simple grammar and prove what the language generated by the grammar actually contains.

Consider the phrase structure grammar $G = \langle \Sigma,\ N,\ S,\ P \rangle$, where:

$$\begin{array}{rcl} \Sigma & = & \{a\} \\ N & = & \{B\} \\ S & = & B \\ P & = & \{B \to \varepsilon \mid aB\} \end{array}$$

Intuitively, this grammar includes all, and nothing but $\mathtt{a}$ sequences. How do we prove this?

Theorem: $\mathcal{L}(G) = \{\mathtt{a}^n | n \geq 0\}$

Proof: Notice that we are trying to prove the equality of two sets. In other words, we want to prove two statements:

1. $\mathcal{L}(G) \subseteq \{\mathtt{a}^n | n \geq 0\}$, or that all sentences are of the form $\mathtt{a}^n$,
2. $\{\mathtt{a}^n | n \geq 0\} \subseteq \mathcal{L}(G)$, or that all strings of the form $\mathtt{a}^n$ are generated by the grammar.

Proof of (1): Looking at the grammar, and using intuition, it is obvious that all sentential forms are of the form: $\mathtt{a}^n B$ or $\mathtt{a}^n$.

This is formally proved by induction on the length of derivation. In other words, we prove that any sentential form derived in one step is of the desired form, and that, if any sentential form derived in k steps takes the given form, so should any sentential form derivable in $k+1$ steps. By induction we then conclude that derivations of any length have the given structure.

Consider $B \overset{1}{\Rightarrow}_G \alpha$. From the grammar, α is either $\varepsilon = \mathtt{a}^0$ or $\mathtt{a}B$, both of which have the desired structure.

Assume that all derivations of length k result in the desired format. Now consider a $k+1$ length relation: $B \overset{k+1}{\Rightarrow}_G \beta$.

But this implies that $B \stackrel{k}{\Rightarrow}_G \alpha \stackrel{1}{\Rightarrow}_G \beta$, where, by induction, α is either of the form $\mathtt{a}^n B$ or $\mathtt{a}^n$. Clearly, we cannot derive any β from $\mathtt{a}^n$, and if $\alpha = \mathtt{a}^n B$ then $\beta = \mathtt{a}^{n+1} B$ or $\beta = \mathtt{a}^{n+1}$ both of which have the desired structure.

Hence, by induction, $\mathcal{S}(G) \subseteq \{\mathtt{a}^n,\ \mathtt{a}^n B \mid n \geq 0\}$. Thus:

$$
\begin{array}{rl}
 & x \in \mathcal{L}(G) \\
\Leftrightarrow & x \in \mathcal{S}(G) \wedge x \in \Sigma^* \\
\Rightarrow & x \in \{\mathtt{a}^n,\ \mathtt{a}^n B \mid n \geq 0\} \wedge x \in \mathtt{a}^* \\
\Leftrightarrow & x \in \mathtt{a}^*
\end{array}
$$

which completes the proof of (1).

Proof of (2): On the other hand, we can show that *all* strings of the form $\mathtt{a}^n B$ are derivable in zero or more steps from B.

The proof once again relies on induction, this time on n.

Base Case ($n = 0$): By definition of zero step derivations $B \stackrel{0}{\Rightarrow}_G B$. Hence $B \stackrel{*}{\Rightarrow}_G B$.

Inductive case: Assume that the property holds for $n = k$: $B \stackrel{*}{\Rightarrow}_G \mathtt{a}^k B$.

But $\mathtt{a}^k B \stackrel{1}{\Rightarrow}_G \mathtt{a}^{k+1} B$, implying that $B \stackrel{*}{\Rightarrow}_G \mathtt{a}^{k+1} B$.

Hence, by induction, for all n, $B \stackrel{*}{\Rightarrow}_G \mathtt{a}^n B$. But $\mathtt{a}^n B \Rightarrow_G \mathtt{a}^n$. By definition of derivations: $B \stackrel{+}{\Rightarrow}_G \mathtt{a}^n$ Thus, if $x = \mathtt{a}^n$ then $x \in \mathcal{L}(G)$:

$$\{\mathtt{a}^n | n \geq 0\} \subseteq \mathcal{L}(G)$$

$\square$

Note: The proof for (1) can be given in a much shorter, neater way. Simply note that $\mathcal{L}(G) \subseteq \Sigma^*$. But $\Sigma = \{\mathtt{a}\}$. Thus, $\mathcal{L}(G) \subseteq \{\mathtt{a}\}^* = \{\mathtt{a}^n \mid n \geq 0\}$. The reason for giving the alternative long proof is to show how induction can be used on the length of derivation.

2.5.2 A More Complex Example

As a more complex example, we will now treat a palindrome generating grammar. To reason formally, we need to define a new operator, the reversal of a string, written as s^R. This is defined by:

$$\begin{array}{rcl} \varepsilon^R & \stackrel{def}{=} & \varepsilon \\ (ax)^R & \stackrel{def}{=} & x^R a \end{array}$$

The set of all palindromes over an alphabet Σ can now be elegantly written as: $\{ww^R \mid w \in \Sigma^*\} \cup \{waw^R \mid a \in \Sigma \wedge w \in \Sigma^*\}$. We will abbreviate this set to Pal_Σ.

The grammar $G' = \langle \Sigma', N', S', P' \rangle$, defined below, should (intuitively) generate exactly the set of palindromes over $\{a, b\}$:

$$\begin{array}{rcl} \Sigma & = & \{a, b\} \\ N & = & \{B\} \\ S & = & B \\ P & = & \{ \; B \to a, \\ & & \quad B \to b, \\ & & \quad B \to \varepsilon, \\ & & \quad B \to aBa, \\ & & \quad B \to bBb \quad \} \end{array}$$

Theorem: The language generated by G' includes all palindromes: $Pal_\Sigma \subseteq \mathcal{L}(G')$.

Proof: If we can prove that all strings of the form wBw^R ($w \in \Sigma^*$) are derivable in one or more steps from S, then, using the production rule $B \to \varepsilon$, we can generate any string from the first set in one or more productions:

$$B \stackrel{*}{\Rightarrow}_{G'} wBw^R \Rightarrow'_G ww^R$$

Similarly, using the rules $B \to a$ and $B \to b$, we can generate any string in the second set.

Now, we must prove that all wBw^R are in sentential form. The proof proceeds by induction on the length of string w.

Base case $|w| = 0$: By definition of $\stackrel{*}{\Rightarrow}_{G'}$, $B \stackrel{*}{\Rightarrow}_{G'} B = \varepsilon B \varepsilon^R$.

Inductive case: Assume that for any terminal string w of length k, $B \stackrel{*}{\Rightarrow}_{G'} wBw^R$.

Given a string w' of length $k+1$, $w' = w\texttt{a}$ or $w' = w\texttt{b}$, for some string w of length k. Hence, by the inductive hypothesis: $B \overset{*}{\Rightarrow}_{G'} wBw^R$.

Consider the case for $w' = w\texttt{a}$. Using the production rule $B \to aBa$:

$$B \overset{*}{\Rightarrow}_{G'} wBw^R \Rightarrow'_G waBaw^R = w'Bw'^R$$

Hence $B \overset{*}{\Rightarrow}_{G'} w'Bw'^R$, completing the proof.

□

Theorem: Furthermore, the grammar generates only palindromes.

Proof: If we now prove that all sentential forms have the structure wBw^R or ww^R or wcw^R (where $w \in \Sigma^*$ and $c \in \Sigma$), recall that $\mathcal{L}(G) = \mathcal{S}(G) \cap \Sigma^*$. Thus, it can then be proved that $\mathcal{L}G \subseteq \{ww^R \mid w \in \Sigma^*\} \cup \{wcw^R \mid c \in \Sigma \wedge w \in \Sigma^*\} = Pal_\Sigma$.

To prove that all sentential forms are in one of the given structures, we use induction on the length of the derivation.

Base case (length 1): If $B \overset{1}{\Rightarrow}_{G'} \alpha$, α takes the form of one of: ε, $\texttt{a}$, $\texttt{b}$, $\texttt{a}B\texttt{a}$, $\texttt{b}B\texttt{b}$, all of which are in the desired form.

Inductive case: Assume that all derivations of length k result in an expression of the desired form.

Now consider a derivation $k+1$ steps long. Clearly, we can split the derivation into two parts, one k steps long, and one last step:

$$B \overset{k}{\Rightarrow}_{G'} \alpha \overset{1}{\Rightarrow}_{G'} \beta$$

Using the inductive hypothesis, α must be in the form of wBw^R or ww^R or wcw^R. The last two are impossible, since otherwise there would be no last step to take (the production rules all transform a non-terminal which is not present in the last two cases). Thus, $\alpha = wBw^R$ for some string of terminals w.

From this α we get only a limited number of last steps, forcing β to be:

- ww^R
- waw^R
- wbw^R

- $waBaw^R = (wa)B(wa)^R$
- $wbBbw^R = (wb)B(wb)^R$

All of which are in the desired format. Thus, any derivation $k+1$ steps long produces a string in one of the given forms. This completes the induction, proving that that all sentential forms have the structure wBw^R, ww^R or wcw^R, which completes the proof.

□

2.5.3 Exercises

1. Consider the following grammar:

   $G = \langle \Sigma,\ N,\ S,\ P \rangle$, where:

$$
\begin{array}{rcl}
\Sigma & = & \{a,\ b\} \\
N & = & \{S,\ A,\ B\} \\
P & = & \{\ \ S \to AB, \\
 & & \quad\ \ A \to \varepsilon \mid aA, \\
 & & \quad\ \ B \to b \mid bB \quad \}
\end{array}
$$

   Prove that $\mathcal{L}(G) = \{a^n b^m | n \geq 0,\ m > 0\}$

2. Consider the following grammar:

   $G = \langle \Sigma,\ N,\ S,\ P \rangle$, where:

$$
\begin{array}{rcl}
\Sigma & = & \{a,\ b\} \\
N & = & \{S,\ A,\ B\} \\
P & = & \{\ \ S \to aB \mid A, \\
 & & \quad\ \ A \to bA \mid S, \\
 & & \quad\ \ B \to bS \mid b \quad \}
\end{array}
$$

   Prove that:

(a) Prove that any string in $\mathcal{L}(G)$ is at least two symbols long.

(b) For any string $x \in \mathcal{L}(G)$, x always ends with a `b`.

(c) The number of occurances of `b` in a sentence in the language generated by G is not less than the number of occurances of `a`.

2.6 Summary

The following points should summarize the contents of this part of the course:

- A language is simply a set of finite strings over an alphabet.
- A grammar is a finite means of deriving a possibly infinite language from a finite set of rules.
- Proofs about languages derived from a grammar usually use induction over one of a number of variables, such as length of derivation, length of string, number of occurances of a symbol in the string, etc.
- The proofs are quite simple and routine once you realize how induction is to be used. The bulk of the time taken to complete a proof is taken up sitting at a table, staring into space and waiting for inspiration. Do not get discouraged if you need to dwell on a problem for a long time to find a solution. Practice should help you speed up inspiration.

Chapter 3

Classes of Languages

3.1 Motivation

The definition of a phrase structure grammar is very general in nature. Implementing a language checking program for a general grammar is not a trivial task and can be very inefficient. This part of the course identifies some classes of languages which we will spend the rest of the course discussing and proving properties of.

3.2 Context Free Languages

3.2.1 Definitions

One natural way of limiting grammars is to allow only production rules which derive a string (of terminals and non-terminals) from a single non-terminal symbol. The basic idea is that in this class of languages, context information (the symbols surrounding a particular non-terminal symbol) does not matter and should not change how a string evolves. Furthermore, once a terminal symbol is reached, it cannot evolve any further. From these restrictions, grammars falling in this class are called context free grammars. An obvious question arising from the definition of this class of languages is: does it in fact reduce the set of languages produced by general grammars? Or conversely, can we construct a context free grammar for any language produced by a phrase structure grammar? The answer is negative. Certain languages, produced by a phrase structure grammar cannot be generated by a context free grammar. An example of such a language is $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$. It is beyond the scope of this introductory course to prove the impossibility of constructing a context free grammar which recognizes this language, however you can try your hand at showing that there is a phrase structure grammar which generates this language.

Definition: A phrase structure grammar $G = \langle \Sigma,\ N,\ P,\ S \rangle$ is said to be a *context free grammar* if all the productions in P are in the form: $A \to \alpha$, where $A \in N$ and $\alpha \in (\Sigma \cup N)^*$.

Definition: A language L is said to be a *context free language* if there is a context free grammar G such that $\mathcal{L}(G) = L$.

Note that the constraints placed on BNF grammars are precisely those placed on context free languages. This gives us an extra incentive to prove properties about this class of languages, since the results we obtain will immediately be applicable to a large number of computer language grammars already defined[1].

3.2.2 Context free languages and the empty string

Productions of the form $\alpha \to \varepsilon$ are called *ε-productions.* It seems like a waste of effort to produce strings which then disappear into thin air! This seems to present one way of limiting context free grammars — by disallowing ε-productions. But are we limiting the set of languages produced?

Definition: A grammar is said to be *ε-free* if it has no ε-productions except possibly for $S \to \varepsilon$ (where S is the start symbol), in which case S does not appear on the right hand side of any rule.

Note that some texts define ε-free to imply no ε-productions at all.

Consider a language which includes the empty string. Clearly, there must be some rule which results in the empty string (possibly $S \to \varepsilon$). Thus,

[1]This is, up to a certain extent, putting the carriage before the horse. When the BNF notation was designed, the basic properties of context free grammars already known. Still, people all around the world continue to define computer language grammars in terms of the BNF notation. Implementing parsers for such grammars is made considerably easier by knowing some basic properties of context free languages.

certain languages cannot, it seems, be produced by a grammar which has no ε-productions. However, as the following results show, the loss is not that great. For any context free language, there is an ε-free context free grammar which generates the language.

Lemma: For any context free grammar with no ε-productions G: $G = \langle\Sigma,\ N,\ P,\ S\rangle$, we can construct a context free grammar G' which is ε-free such that $\mathcal{L}(G') = \mathcal{L}(G) \cup \{\varepsilon\}$.

Strategy: To prove the lemma we construct grammar G'. The new grammar is identical to G except that it starts off from a new start S' for which there are two new production rules. $S' \to \varepsilon$ produces the desired empty string and $S' \to S$ guarantees that we also generate all the strings in G. G' is obviously ε free and it is intuitively obvious that it also generates $\mathcal{L}(G)$.

Proof: Define grammar $G' = \langle\Sigma,\ N',\ P',\ S'\rangle$ such that

$$\begin{aligned} N' &= N \cup \{S'\} \text{ (where } S' \notin N) \\ P' &= P \cup \{S' \to \varepsilon,\ S' \to S\} \end{aligned}$$

Clearly, G' satisfies the constraints that it is an ε free context free grammar since G itself is an ε free context free grammar. We thus need to prove that $\mathcal{L}(G') = \mathcal{L}(G) \cup \{\varepsilon\}$.

Part 1: $\mathcal{L}(G') \subseteq \mathcal{L}(G) \cup \{\varepsilon\}$.

Consider $x \in \mathcal{L}(G')$. By definition, $S' \overset{+}{\Rightarrow}_{G'} x$ and $x \in \Sigma^*$.

By definition, $S' \overset{+}{\Rightarrow}_{G'} x$ if, either:

- $S' \overset{1}{\Rightarrow}_{G'} x$, which by case analysis of P' and the condition $x \in \Sigma^*$, implies that $x = \varepsilon$. Hence $x \in \mathcal{L}(G) \cup \{\varepsilon\}$.

- $S' \overset{n+1}{\Rightarrow}_{G'} x$, where $n \geq 1$. This in turn implies that:

  $$S' \overset{1}{\Rightarrow}_{G'} \alpha \overset{n}{\Rightarrow}_{G'} x$$

  By case analysis of P', $\alpha = S$. Furthermore, in the derivation $S \overset{n}{\Rightarrow}_{G'} x$, S' does not appear (can be checked by induction on length of derivation). Hence, it uses only production rules in P which guarantees that $S \overset{n}{\Rightarrow}_{G} x$ ($n \geq 1$).

Thus $x \in \mathcal{L}(G) \cup \{\varepsilon\}$.

Hence, $x \in \mathcal{L}(G') \Rightarrow x \in \mathcal{L}(G) \cup \{\varepsilon\}$, which is what is required to prove that $\mathcal{L}(G') \subseteq \mathcal{L}(G) \cup \{\varepsilon\}$.

Part 2: $\mathcal{L}(G) \cup \{\varepsilon\} \subseteq \mathcal{L}(G')$

This result follows similarly. If $x \in \mathcal{L}(G) \cup \{\varepsilon\}$, then either:

- $x \in \mathcal{L}(G)$ implying that $S \overset{+}{\Rightarrow}_G x$. But, since $P \subseteq P'$, we can deduce that:

  $$S' \Rightarrow'_G S \overset{+}{\Rightarrow}_{G'} x$$

  Implying that $x \in \mathcal{L}(G')$.

- $x \in \{\varepsilon\}$ implies that $x = \varepsilon$. From the definition of G', $S' \Rightarrow'_G \varepsilon$, hence $\varepsilon \in \mathcal{L}(G')$.

Hence, in both cases, $x \in \mathcal{L}(G')$, completing the proof.

$\square$

Example: Given the following grammar G, produce a new grammar G' which satisfies $\mathcal{L}(G) \cup \{\varepsilon\} = \mathcal{L}(G')$. $G = \langle \Sigma,\ N,\ P,\ S \rangle$ where:

$$\begin{array}{rcl}
\Sigma & = & \{a,\ b\} \\
N & = & \{S,\ A,\ B\} \\
P & = & \{ \quad S \to A \mid B \mid AB, \\
 & & \quad\quad A \to a \mid aA, \\
 & & \quad\quad B \to b \mid bB \qquad\qquad \}
\end{array}$$

Using the method used in the lemma just proved, we can write $G' = \langle \Sigma,\ N \cup \{S'\},\ P',\ S' \rangle$, where $P' = P \cup \{S' \to S \mid \varepsilon\}$, which is guaranteed to satisfy the desired property.

$$
\begin{array}{l}
G' = \langle \Sigma,\ N',\ P',\ S' \rangle \\
\qquad \begin{array}{rcl}
N' & = & \{S',\ S,\ A,\ B\} \\
P' & = & \{ \quad S' \to \varepsilon \mid S, \\
 & & \qquad S \to A \mid B \mid AB, \\
 & & \qquad A \to a \mid aA, \\
 & & \qquad B \to b \mid bB \qquad \}
\end{array}
\end{array}
$$

Theorem: For any context free grammar $G = \langle \Sigma,\ N,\ P,\ S \rangle$, we can construct a context free grammar G' with no ε-productions such that $\mathcal{L}(G') = \mathcal{L}(G) \setminus \{\varepsilon\}$.

Strategy: Again we construct grammar G' to prove the claim. The strategy we use is as follows:

- we copy all non-ε-productions from G to G'.
- for any non-terminal N which can become ε, we copy every rule in which N appears on the right hand side both with and without N.

Thus, for example, if $A \stackrel{+}{\Rightarrow}_G \varepsilon$, and there is a rule $B \to AaA$ in P, then we add productions $B \to Aa$, $B \to aA$, $B \to AaA$ and $B \to a$ to P'.

Clearly G' satisfies the property of having no ε-productions, as required. However, the proof of equivalence (modulo ε) of the two languages is still required.

Proof: Define $G' = \langle \Sigma,\ N,\ P',\ S \rangle$, where P' is defined to be the union of the following sets of production rules:

- $\{A \to \alpha \mid \alpha \neq \varepsilon,\ A \to \alpha \in P\}$ — all non-ε-production rules in P.
- If N_ε is defined to be the set of all non-terminal symbols from which ε can be derived ($N_\varepsilon = \{A \mid A \in N,\ A \stackrel{*}{\Rightarrow}_G \varepsilon\}$), then we take the production rules in P and remove arbitrarily any number of non-terminals which are in N_ε (making sure we do not end up with a ε-production).

By definition, G' is ε-free. What is still left to prove is that $\mathcal{L}(G') = \mathcal{L}(G) \setminus \{\varepsilon\}$.

It suffices to prove that: For every $x \in \Sigma^* \setminus \{\varepsilon\}$, $S \stackrel{+}{\Rightarrow}_G x$ if and only if $S \stackrel{+}{\Rightarrow}_{G'} x$ and that $\varepsilon \notin \mathcal{L}(G')$.

To prove the second statement we simply note that, to produce ε, the last production must be an ε-production, of which G' does not have any.

To prove the first, we start by showing that for every non-terminal A, $x \in \Sigma^* \setminus \{\varepsilon\}$, $A \stackrel{+}{\Rightarrow}_G x$ if and only if $A \stackrel{+}{\Rightarrow}_{G'} x$. The desired result is simply a special case ($A = S$) which would then complete the proof of the theorem.

Proof that $A \stackrel{+}{\Rightarrow}_G x$ implies $A \stackrel{+}{\Rightarrow}_{G'} x$:

Proof by strong induction on the length of the derivation.

Base case: $A \stackrel{1}{\Rightarrow}_G x$. Thus, $A \rightarrow x \in P$ and also in P' (since it is not an ε-production). Hence, $A \stackrel{+}{\Rightarrow}_{G'} x$.

Assume it holds for any production taking up to k steps. We now need to prove that $A \stackrel{k+1}{\Rightarrow}_G x$ implies $A \stackrel{+}{\Rightarrow}_{G'} x$.

But if $A \stackrel{k+1}{\Rightarrow}_G x$ then $A \stackrel{1}{\Rightarrow}_G X_1 \ldots X_n \stackrel{k}{\Rightarrow}_G x$. x can also be split into n parts ($x = x_1 \ldots x_n$, some of which may be ε) such that $X_i \stackrel{*}{\Rightarrow}_G x_i$.

Now consider all non-empty x_i: $x = x_{\lambda_1} \ldots x_{\lambda_m}$. Since the productions of these strings all take up to k steps, we can deduce from the inductive hypothesis that: $X_{\lambda_i} \stackrel{+}{\Rightarrow}_{G'} x_{\lambda_i}$. Since all the remaining non-terminals can produce ε, we have a production rule (of the second type) in P': $A \rightarrow X_{\lambda_1} \ldots X_{\lambda_m}$.

Hence $A \Rightarrow'_G X_{\lambda_1} \ldots X_{\lambda_m} \stackrel{+}{\Rightarrow}_{G'} x_{\lambda_1} \ldots x_{\lambda_m} = x$, completing the induction.

Since all such productions take up to k steps, we can deduce from the inductive principle that: $X_i \stackrel{+}{\Rightarrow}_{G'} x_i$ for every non-empty x_i.

Proof that $A \stackrel{+}{\Rightarrow}_{G'} x$ implies $A \stackrel{+}{\Rightarrow}_G x$:

$\square$

Corollary: For any context free grammar G, we can construct an ε-free context free grammar G' such that $\mathcal{L}(G') = \mathcal{L}(G)$.

Proof: The result follows immediately from the lemma and theorem just proved. From G, we can construct an context free grammar G'' with no ε-productions such that $\mathcal{L}(G'') = \mathcal{L}(G) \setminus \{\varepsilon\}$ (by theorem).

Now, if $\varepsilon \notin \mathcal{L}(G)$, we have $\mathcal{L}(G'') = \mathcal{L}(G)$, hence G' is defined to be G''. Note that G'' contains no ε-productions and is thus ε-free.

If $\varepsilon \in \mathcal{L}(G)$, using the lemma, we can produce a grammar G' from G'' such that $\mathcal{L}(G') = \mathcal{L}(G'') \cup \{\epsilon\}$, where G' is ε-free. It can now be easily shown that $\mathcal{L}(G') = \mathcal{L}(G)$.

$\square$

Example: Construct an ε-free context free grammar and which generates the same language as $G = \langle \Sigma,\ N,\ P,\ S \rangle$:

$$
\begin{array}{rcl}
\Sigma & = & \{a,\ b\} \\
N & = & \{S,\ A,\ B\} \\
P & = & \{\ \ S \to aA \mid bB \mid AabB, \\
 & & \quad\ \ A \to \varepsilon \mid aA, \\
 & & \quad\ \ B \to \varepsilon \mid bS \qquad\qquad \}
\end{array}
$$

Using the method as described in the theorem, we construct a context free G' with no ε-productions such that $\mathcal{L}(G') = \mathcal{L}(G) \setminus \{\varepsilon\}$. From the theorem, $G' = \langle \Sigma,\ N,\ P',\ S \rangle$, where P' is the union of:

- The productions in P which are not ε-productions:

  $$\{S \to aA \mid bB \mid AabB,\ A \to aA,\ B \to bS\}$$

- Of the three non-terminals, $A \stackrel{+}{\Rightarrow}_G \varepsilon$ and $B \stackrel{+}{\Rightarrow}_G \varepsilon$, but ε cannot be derived from S (S immediately produces a terminal symbol which cannot disappear since G is a context free grammar). We now rewrite all the rules in P leaving out combinations of A and B:

  $$\{S \to a \mid b \mid Aab \mid abB,\ A \to a\}$$

From the result of the theorem, $\mathcal{L}(G') = \mathcal{L}(G) \setminus \{\varepsilon\}$. But $\varepsilon \notin \mathcal{L}(G)$. Hence, $\mathcal{L}(G) \setminus \{\varepsilon\} = \mathcal{L}(G)$. The result we need is G':

$$
\begin{array}{l}
G' = \langle \Sigma,\ N,\ P',\ S \rangle \\
\qquad\qquad\qquad \begin{array}{rcl}
\Sigma & = & \{a,\ b\} \\
N & = & \{S,\ A,\ B\} \\
P' & = & \{\ \begin{array}{ll}
S \rightarrow & a \mid b \mid aA \mid bB \mid \\
 & abB \mid Aab \mid AabB, \\
A \rightarrow & a \mid aA, \\
B \rightarrow & bS \qquad\qquad\qquad \}
\end{array}
\end{array}
\end{array}
$$

Example: Construct an ε-free context free grammar and which generates the same language as $G = \langle \Sigma,\ N,\ P,\ S \rangle$:

$$
\begin{array}{rcl}
\Sigma & = & \{a,\ b\} \\
N & = & \{S,\ A,\ B\} \\
P & = & \{\ \begin{array}{l}
S \rightarrow A \mid B \mid ABa, \\
A \rightarrow \varepsilon \mid aA, \\
B \rightarrow bS \qquad\qquad \}
\end{array}
\end{array}
$$

Using the theorem just proved, we will first define G' which satisfies $\mathcal{L}(G') = \mathcal{L}(G) \setminus \{\varepsilon\}$.

$G' \stackrel{def}{=} \langle \Sigma,\ N,\ P',\ S \rangle$ where P' is defined to be the union of:

- The non-ε-producing rules in P: $\{S \rightarrow A \mid B \mid ABa,\ A \rightarrow aA,\ B \rightarrow bS\}$.
- The non-terminal symbols which can produce ε are A, S ($S \Rightarrow_G A \Rightarrow_G \varepsilon$). Clearly B cannot produce ε. We now add all rules in P leaving out instances of A and S (which, in the process do not become ε-productions): $\{S \rightarrow Ba,\ A \rightarrow a,\ B \rightarrow b\}$.

$$
\begin{array}{rcl}
P' & = & \{\ \begin{array}{l}
S \rightarrow A \mid B \mid ABa \mid Ba, \\
A \rightarrow a \mid aA, \\
B \rightarrow b \mid bS \qquad\qquad \}
\end{array}
\end{array}
$$

However, $\varepsilon \in \mathcal{L}(G)$. We thus need to produce a context free grammar G'' whose language is exactly that of G' together with ε. Using the method from the lemma, we get $G'' = \Sigma,\ N \cup \{S'\},\ P'',\ S'\rangle$ where $P'' = P' \cup \{S \to \varepsilon \mid S\}$.

By the result of the theorem and lemma, G'' is the grammar requested.

3.2.3 Derivation Order and Ambiguity

It has already been noted in the first set of exercises, that certain context free grammars are ambiguous, in the sense that for certain strings, more than one derivation tree is possible.

The syntax tree is constructed as follows:

1. Draw the root of the tree with the initial non-terminal S written inside it.
2. Choose one leaf node with any non-terminal A written inside it.
3. Use any production rule (once) to derive a string α from A.
4. Add a child node to A for every symbol (terminal or non-terminal) in α, such that the children would read left-to-right α.
5. If there are any non-terminal leaves left, jump back to instruction 2.

Reading the terminal symbols from left to right gives the derived string x. The tree is called the syntax tree of x.

The sequence of production rules as used in the construction of the syntax tree of x corresponds to a particular derivation of x from S. If the intermediate trees during the construction read (as before, left to right) α_1, α_2, $\ldots \alpha_n$, this would correspond to the derivation $S \Rightarrow_G \alpha_1 \Rightarrow_G \ldots \Rightarrow_G \alpha_n \Rightarrow_G x$. As the forthcoming examples show, different syntax trees correspond to different derivations, but different derivations may have a common syntax tree.

For example, consider the following grammar:

$$G \stackrel{def}{=} \langle\{\mathtt{a},\mathtt{b}\},\{S,A\},P,S\rangle$$

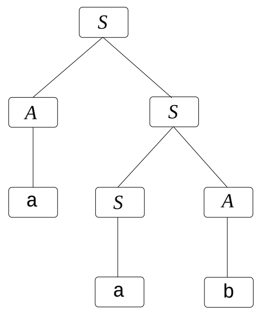
$$P \stackrel{def}{=} \{S \to SA \mid AS \mid \mathtt{a},\ A \to \mathtt{a} \mid \mathtt{b}\}$$

Now consider the following two possible derivations of `aab`:

$$
\begin{aligned}
S &\Rightarrow_G AS \\
&\Rightarrow_G ASA \\
&\Rightarrow_G AS\mathtt{b} \\
&\Rightarrow_G \mathtt{a}S\mathtt{b} \\
&\Rightarrow_G \mathtt{aab}
\end{aligned}
$$

$$
\begin{aligned}
S &\Rightarrow_G AS \\
&\Rightarrow_G ASA \\
&\Rightarrow_G \mathtt{a}SA \\
&\Rightarrow_G \mathtt{a}S\mathtt{b} \\
&\Rightarrow_G \mathtt{aab}
\end{aligned}
$$

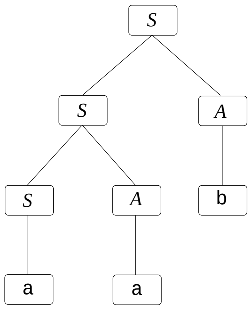
If we draw the derivation trees for both, we discover that they are, in fact equivalent:



However, now consider another derivation of `aab`:

$$
\begin{aligned}
S &\Rightarrow_G SA \\
&\Rightarrow_G SAA \\
&\Rightarrow_G \mathtt{a}AA \\
&\Rightarrow_G \mathtt{a}A\mathtt{b} \\
&\Rightarrow_G \mathtt{aab}
\end{aligned}
$$

This has a different parse tree:



Hence G is an ambiguous grammar.

Thus, every syntax tree can be followed in a multitude of ways (as shown in the previous example). A derivation is said to be a leftmost derivation if all derivation steps are done on the first (leftmost) non-terminal. Any tree thus corresponds to exactly one leftmost derivation.

The leftmost derivation related to the first syntax tree of x is:

$$\begin{aligned} S &\Rightarrow_G AS \\ &\Rightarrow_G \mathtt{a}S \\ &\Rightarrow_G \mathtt{a}SA \\ &\Rightarrow_G \mathtt{aa}A \\ &\Rightarrow_G \mathtt{aab} \end{aligned}$$

while the leftmost derivation related to the second syntax tree of x is:

$$\begin{aligned} S &\Rightarrow_G SA \\ &\Rightarrow_G SAA \\ &\Rightarrow_G \mathtt{a}AA \\ &\Rightarrow_G \mathtt{aa}A \\ &\Rightarrow_G \mathtt{aab} \end{aligned}$$

Since every syntax tree can be traversed left to right, and every leftmost derivation has a syntax tree, we can say that a grammar is ambiguous if there is a string x which has at least 2 distinct leftmost derivations.

3.2.4 Exercises

1. Given the context free grammar G:

$$
G = \langle \Sigma,\ N,\ P,\ S \rangle
$$

$$
\begin{array}{rcl}
\Sigma & = & \{a,\ b\} \\
N & = & \{S,\ A,\ B\} \\
P & = & \{\ \begin{array}{ll} S \rightarrow & AA \mid BB, \\ A \rightarrow & a \mid aA, \\ B \rightarrow & b \mid bB \end{array} \quad \}
\end{array}
$$

   (a) Describe $\mathcal{L}(G)$.
   (b) Formally prove that $\varepsilon \notin \mathcal{L}(G)$.
   (c) Give a context free grammar G' such that $\mathcal{L}(G') = \mathcal{L}(G) \cup \{\varepsilon\}$.

2. Given the context free grammar G:

$$
G = \langle \Sigma,\ N,\ P,\ S \rangle
$$

$$
\begin{array}{rcl}
\Sigma & = & \{a,\ b,\ c\} \\
N & = & \{S,\ A,\ B\} \\
P & = & \{\ \begin{array}{ll} S \rightarrow & A \mid B \mid cABc, \\ A \rightarrow & \varepsilon \mid aS, \\ B \rightarrow & \varepsilon \mid bS \end{array} \quad \}
\end{array}
$$

   (a) Describe $\mathcal{L}(G)$.
   (b) Formally prove whether ε is in $\mathcal{L}(G)$.
   (c) Define an ε-free context free grammar G' satisfying $\mathcal{L}(G') = \mathcal{L}(G)$.

3. Given the context free grammar G:

$$
G = \langle \Sigma,\ N,\ P,\ S \rangle
$$

$$
\Sigma \ = \ \{a,\ b,\ c\}
$$

$$
\begin{array}{rcll}
N & = & \{S,\ A,\ B,\ C\} & \\
P & = & \{ \quad S \rightarrow & AC \mid CB \mid cABc, \\
 & & \quad A \rightarrow & \varepsilon \mid aS, \\
 & & \quad B \rightarrow & \varepsilon \mid bS \\
 & & \quad C \rightarrow & c \mid cc \qquad\qquad \}
\end{array}
$$

(a) Formally prove whether ε is in $\mathcal{L}(G)$.

(b) Define an ε-free context free grammar G' satisfying $\mathcal{L}(G') = \mathcal{L}(G)$.

4. Give four distinct leftmost derivations of `aaa` in the grammar G defined below. Draw the syntax trees for these derivations.

$$G \stackrel{def}{=} \langle\{\mathtt{a}, \mathtt{b}\}, \{S, A\}, P, S\rangle$$

$$P \stackrel{def}{=} \{S \rightarrow SA \mid AS \mid \mathtt{a},\ A \rightarrow \mathtt{a} \mid \mathtt{b}\}$$

5. Show that the grammar below is ambiguous:

$$G \stackrel{def}{=} \langle\{\mathtt{a}, \mathtt{b}\}, \{S, A, B\}, P, S\rangle$$

$$
\begin{array}{rcll}
P & \stackrel{def}{=} & \{ & S \rightarrow SA \mid AS \mid B, \\
 & & & A \rightarrow \mathtt{a} \mid \mathtt{b}, \\
 & & & B \rightarrow \mathtt{b} \qquad\qquad \}
\end{array}
$$

3.3 Regular Languages

Context free grammars are a convenient step down from general phrase structure grammars. The popularity of the BNF notation indicates that these grammars are generally enough to describe the syntactic structure of general programming languages. Furthermore, as we will see later, computationwise, these grammars can be conveniently parsed.

However, using the properties of context free grammars for certain languages is like cracking a nut with a sledgehammer. If we were to define a simpler subset of grammars which is still general enough to include these smaller languages, we would have stronger results about this subset than we would have about context free grammars, which might mean more efficient parsing.

Context free grammars limit what can appear on the left hand side of a parse rule to a bare minimum (a single non-terminal symbol). If we are to simplify grammars by placing further constraints on the production rules it has to be on the right hand side of these rules. Regular grammars place exactly such a constraint: Every rule must produce either a single terminal symbol, or a single terminal symbol followed by a single non-terminal.

The constraints on the left hand side of production rules is kept the same, implying that every regular grammar is also a context free one. Again, we have to ask the question: is every context free language also expressible by a regular grammar? In other words, is the class of context free languages just the same as the class of regular languages? The answer is once again negative. $\{a^n b^n \mid n \geq 0\}$ is a context free language (find the grammar which generates it) but not a regular grammar.

3.3.1 Definitions

Definition: A phrase structure grammar $G = \langle \Sigma,\ N,\ P,\ S \rangle$ is said to be a *regular grammar* if all the productions in P are in on of the following forms:

- $A \to \varepsilon$, where $A \in N$
- $A \to a$, where $A \in N$ and $a \in \Sigma$
- $A \to aB$, where $A,\ B \in N$ and $a \in \Sigma$

Note: This definition does not exclude multiple rules for a single non-terminal. Recall that, for example, $A \to a \mid aA$ is shorthand for the two production rules $A \to a$ and $A \to aA$. Both these rules are allowed in regular grammars, and thus so is $A \to a \mid aA$.

Definition: A language L is said to be a *regular language* if there is a regular grammar G such that $\mathcal{L}(G) = L$.

3.3.2 Properties of Regular Grammars

Proposition: Every regular language is also a context free language.

Proof: If L is a regular language, there is a regular grammar G which generates it. But every production rule in G has the form $A \to \alpha$ where $A \in N$. Thus, G is a context free grammar. Since G generates L, L is a context free language.

$\square$

Proposition: If G is a regular grammar, every sentential form of G contains at most one non-terminal symbol. Furthermore, the non-terminal will always be the last symbol in the string.

$$\mathcal{S}(G) \subseteq \Sigma^* \cup \Sigma^* N$$

Again, we are interested whether, for any regular language L, there always exists an equivalent ε-free regular language. We will try to follow the same strategy as used with context free grammars.

Proposition: For every regular grammar G, there exists a regular grammar G' with no ε-productions such that $\mathcal{L}(G') = \mathcal{L}(G) \setminus \{\varepsilon\}$.

Proof: We will use the same construction as for context free grammars. Recall that the construction used in that theorem removed all ε-productions and copied all rules leaving out all combinations of non-terminals which can produce the ε. Note that the only rules in regular grammars with non-terminals on the right-hand side are of the form $A \to aB$. Leaving out B gives the production $A \to a$ which is acceptable in a regular grammar. The same construction thus yields a regular grammar with no ε-productions. Since we have already proved that the grammar constructed in this manner accepts all strings accepted by the original grammar except for ε, the proof is completed.

$\square$

Proposition: Given a regular grammar G with no ε-productions, we can construct an ε-free regular grammar G' such that $\mathcal{L}(G') = \mathcal{L}(G) \cup \{\varepsilon\}$.

Strategy: With context free grammars we simply added the new productions $S' \to \varepsilon \mid S$. However note that $S' \to S$ is not a valid regular grammar production. What can be done? Note that after using this production, any derivation will need to follow a rule from S. Thus we can replace it by the family of rules $S' \to \alpha$ such that $S \to \alpha$ was in the original grammar.

$$
\begin{array}{l}
G' \stackrel{def}{=} \langle \Sigma,\ N',\ P',\ S' \rangle \\
\qquad \begin{array}{rcl}
N' & \stackrel{def}{=} & N \cup \{S'\} \\
P' & \stackrel{def}{=} & \quad \{S' \to \alpha \mid S \to \alpha \in P\} \\
 & & \cup \ \ \{S' \to \varepsilon\} \\
 & & \cup \ \ P
\end{array}
\end{array}
$$

It is not difficult to prove the equivalence between the two grammars.

□

Theorem: For every regular grammar G, there exists an equivalent ε-free regular grammar G'.

Proof: We start by producing a regular grammar G'' with no ε-productions such that $\mathcal{L}(G'') = \mathcal{L}(G) \setminus \{\varepsilon\}$ as done in the first proposition.

Clearly, if ε was not in the original language $\mathcal{L}(G)$ we now have an equivalent ε-free regular grammar. If it was we use the construction in the second proposition to add ε to the language.

□

Thus, in future discussions, we can freely discuss ε-free regular languages without having limited the scope of the discourse.

Example: Consider the regular grammar G:

$$
\begin{array}{l}
G \stackrel{def}{=} \langle \Sigma,\ N,\ P,\ S \rangle \\
\qquad \begin{array}{rcl}
\Sigma & \stackrel{def}{=} & \{a,\ b\} \\
N & \stackrel{def}{=} & \{S,\ A,\ B\} \\
P & \stackrel{def}{=} & \{\ \ S \to aB \mid bA \\
 & & \quad\ A \to b \mid bS \\
 & & \quad\ B \to a \mid aS \quad \}
\end{array}
\end{array}
$$

Construct a regular grammar G' such that $\mathcal{L}(G) \cup \{\varepsilon\} = \mathcal{L}(G')$.

Using the method prescribed in the proposition, we construct a grammar G' which starts off from a new state S', and can do everything G can do plus evolve from S' to the empty string ($S' \rightarrow \varepsilon$) or to anything S can evolve to ($S' \rightarrow aB \mid bA$):

$$
\begin{array}{l}
G' \stackrel{def}{=} \langle \Sigma,\ N',\ P',\ S' \rangle \\
\qquad\qquad\qquad \begin{array}{rcl}
N' & \stackrel{def}{=} & \{S',\ S,\ A,\ B\} \\
P' & \stackrel{def}{=} & \{\ \ \begin{array}{l} S' \rightarrow aB \mid bA \mid \varepsilon \\ S \rightarrow aB \mid bA \\ A \rightarrow b \mid bS \\ B \rightarrow a \mid aS \qquad\qquad \} \end{array}
\end{array}
\end{array}
$$

3.3.3 Exercises

1. Write a regular grammar with alphabet `a`, `b`, to generate all possible strings over that alphabet which do not include the substring `aaa`.

2. Construct a regular grammar which recognizes exactly all strings from the language $\{a^n b^m \mid n \geq 0,\ m \geq 0,\ n + m > 0\}$. From your grammar derive a new grammar which accepts the language $\{a^n b^m \mid n \geq 0,\ m \geq 0\}$.

3. Prove that in the language generated by G, as defined below, contains only strings with the same number of `a`s and `b`s.

$$
\begin{array}{l}
G \stackrel{def}{=} \langle \Sigma,\ N,\ P,\ S \rangle \\
\qquad\qquad\qquad \begin{array}{rcl}
\Sigma & \stackrel{def}{=} & \{a,\ b\} \\
N & \stackrel{def}{=} & \{S,\ A,\ B\} \\
P & \stackrel{def}{=} & \{\ \ \begin{array}{l} S \rightarrow bB \mid aA \\ A \rightarrow b \mid bS \\ B \rightarrow a \mid aS \qquad \} \end{array}
\end{array}
\end{array}
$$

Also prove that all sentences are of even length.

4. Give a regular grammar (not necessarily ε-free) to show that just adding the rule $S \to \varepsilon$ (S begin the start symbol) does not always yield a grammar which accepts the original language together with ε.

5. An easy test to prove whether $\varepsilon \in \mathcal{L}(G)$ is to use the equivalence: $\varepsilon \in \mathcal{L}(G) \Leftrightarrow S \to \varepsilon \in P$, where S is the start symbol and P is the set of productions.

   Prove the above equivalence.

   Hint: One direction is trivial. For the other, assume that $S \to \varepsilon \notin P$ and prove that $S \stackrel{+}{\Rightarrow}_G \alpha$ would imply that $|\alpha| > 0$.

   Why doesn't this method work with context free grammars in general?

3.3.4 Properties of Regular Languages

The language classes enjoy certain properties, some of which will be found useful in proofs presented later in the course. This part of the course also helps to increase exposure to inductive proof techniques.

Both classes of languages defined so far are closed under union, catenation and Kleene closure. In other words, for example, given two regular languages L_1 and L_2, their union $L_1 \cup L_2$, their catenation $L_1 L_2$ and their Kleene closure ${L_1}^*$ are also regular languages. Similarly for two context free languages.

Here we will prove the result for regular languages since we will need it later in the course. Closure of context free grammars will be treated in the second year course CSM 206 *Language Hierarchies and Algorithmic Complexity*.

The proofs are all by construction, which is another reason for their discussion. After understanding these proofs, for example, anyone can go out and mechanically construct a regular grammar accepting strings in either of two languages already specified using a regular grammar.

Theorem: If L is a regular grammar, so is L^*.

Strategy: The idea is to copy the production rules of a regular grammar which generates L and, for every rule in the form $A \to a$ ($A \in N$, $a \in \Sigma$) add also the rule $A \to aS$, where S is the start symbol. Since S now appears on the right hand side of some rules, we leave out $S \to \varepsilon$.

This way we have a grammar to generate $L^* \setminus \{\varepsilon\}$. Since $\varepsilon \in L^*$, we then add ε by using the lemma in section 3.3.2.

Proof: Since L is a regular language, there must be a regular grammar G such that $L = \mathcal{L}(G)$. We start off by defining a regular grammar G^+ which generates $L^* \setminus \{\varepsilon\}$.

Since $\varepsilon \in L^*$, we then use the lemma given in section 3.3.2 to construct a regular grammar G^* from G^+ such that:

$$\begin{aligned} \mathcal{L}(G^*) &= \mathcal{L}(G^+) \cup \{\varepsilon\} \\ &= (L^* \setminus \{\varepsilon\}) \cup \{\varepsilon\} \\ &= L^* \end{aligned}$$

which will complete the proof.

If $G = \langle \Sigma,\ N,\ P,\ S \rangle$, we define G^+ as follows:

$$\begin{aligned} G^+ &\stackrel{def}{=} \langle \Sigma,\ N,\ P^+,\ S \rangle \\ P^+ &\stackrel{def}{=} \begin{array}{ll} & P \setminus \{S \to \varepsilon\} \\ \cup & \{A \to aS \mid A \to a \in P,\ A \in N,\ a \in \Sigma\} \end{array} \end{aligned}$$

We now need to show that $\mathcal{L}(G^+) = \mathcal{L}(G)^* \setminus \{\varepsilon\}$.

Proof of part 1: $\mathcal{L}(G^+) \subseteq \mathcal{L}(G)^* \setminus \{\varepsilon\}$

Assume that $x \in \mathcal{L}(G^+)$. We prove by induction on the number of times that the new rules $(P^+ \setminus P)$ were used in the derivation of x in G^+.

Base case: If zero applications of the new rules appeared in $S \stackrel{+}{\Rightarrow}_{G^+} x$, $S \stackrel{+}{\Rightarrow}_G x$. Furthermore, there are no rules in G^+ which allow x to be ε. Hence, $\mathcal{L}(G)^* \setminus \{\varepsilon\}$.

Inductive case: Assume that the result holds for derivations using the new rules k times. Now consider x such that $S \stackrel{+}{\Rightarrow}_{G^+} x$, where the new rules have been used $k+1$ times.

If the last new rule used was $A \to aS$, we can rewrite the derivation as:

$$S \stackrel{+}{\Rightarrow}_{G^+} sA \Rightarrow^+_G saS \stackrel{+}{\Rightarrow}_{G^+} saz = x$$

(Recall that all non-terminal sentential forms in a regular language must be of the form sA where $s \in \Sigma^*$ and $A \in N$)

Since no new rules have been used in the last part of the derivation: $saS \stackrel{+}{\Rightarrow}_G saz = x$. Since all rules are applied exclusively to non-terminals: $S \stackrel{+}{\Rightarrow}_G z$

Furthermore, $S \stackrel{+}{\Rightarrow}_{G^+} sA \Rightarrow_G sa$ is a valid derivation which uses only k occurances of the new rules. Hence, by the inductive hypothesis: $sa \in \mathcal{L}(G)^* \subseteq \{\varepsilon\}$.

Since $x = saz$ $x \in \mathcal{L}(G)(\mathcal{L}(G)^* \subseteq \{\varepsilon\})$ which implies that $x \in \mathcal{L}(G)^* \subseteq \{\varepsilon\}$, completing the inductive proof.

Proof of part 2: $\mathcal{L}(G)^* \setminus \{\varepsilon\} \subseteq \mathcal{L}(G^+)$

Assume $x \in \mathcal{L}(G)^* \setminus \{\varepsilon\}$, then $x \in \mathcal{L}(G)^n$ for some value of $n \geq 1$. We prove that $x \in \mathcal{L}(G^+)$ by induction on n.

Base case ($n = 1$): $x \in \mathcal{L}(G)$. Thus $S \stackrel{+}{\Rightarrow}_G x$. But since all the rules of G are in G^+, $S \stackrel{+}{\Rightarrow}_{G^+} x$. Hence $x \in \mathcal{L}(G^+)$. Note that if $S \to \varepsilon$ appeared in P, S could not appear on the right hand side of rules, and hence if it was used in the derivation of x, it must have been used immediately, implying that $x = \varepsilon$.

Inductive case: Assume that all strings in $\mathcal{L}(G)^k$ are in $\mathcal{L}(G^+)$. Now consider $x \in \mathcal{L}(G)^{k+1}$. By definition, $x = st$ where $s \in \mathcal{L}(G)^k$ and $t \in \mathcal{L}(G)$.

By the induction hypothesis, $s \in \mathcal{L}(G^+)$ which means that $S \stackrel{+}{\Rightarrow}_{G^+} s$. But if the last rule applied was $A \to a$: $S \stackrel{*}{\Rightarrow}_{G^+} wA \Rightarrow_G^+ wa = s$.

$$S \stackrel{*}{\Rightarrow}_{G^+} wA \Rightarrow_G^+ waS \stackrel{+}{\Rightarrow}_{G^+} wat = st$$

Hence $x = st \in \mathcal{L}(G^+)$, completing the induction.

$\square$

Example: Consider grammar G which generates all sequences of $\mathtt{a}$ sandwiched between an initial and final $\mathtt{b}$:

$$G = \langle \{\mathtt{a}, \mathtt{b}\}, \{S, A\}, P.S \rangle$$

$$P \;=\; \{S \to \mathtt{b}A,\ A \to \mathtt{a}A \mid \mathtt{b}\}$$

To construct a new regular grammar G^* such that $\mathcal{L}(G^*) = (\mathcal{L}(G))^*$, we apply the method used in Kleene's theorem: we first copy all productions from P to P' and, for every production of the form $A \to a$ in P (a is a terminal symbol), we also add the production $A \to aA$ to P', obtaining grammar G' in the process:

$$G' = \langle\{\mathtt{a}, \mathtt{b}\}, \{S, A\}, P'.S\rangle$$
$$P \;=\; \{S \to \mathtt{b}A,\ A \to \mathtt{a}A \mid \mathtt{b} \mid \mathtt{b}S\}$$

Now we add ε to the language of G' to obtain G^*:

$$G^* = \langle\{\mathtt{a}, \mathtt{b}\}, \{S^*, S, A\}, P'.S^*\rangle$$
$$\begin{array}{rcll} P & = & \{ & S^* \to \mathtt{b}A \mid \varepsilon \\ & & & S \to \mathtt{b}A, \\ & & & A \to \mathtt{a}A \mid \mathtt{b} \mid \mathtt{b}S \quad \} \end{array}$$

Theorem: If L_1 and L_2 are both regular languages, then so is their catenation L_1L_2.

Strategy: We do not give a proof but just the construction. The proof is not too difficult and you should find it good practice!

Let L_i be generated by regular grammar $G_i = \langle\Sigma_i,\ N_i,\ P_i,\ S_i\rangle$. We assume that the non-terminal symbols are disjoint, otherwise we rename them. The idea is to start off from the start symbol of G_1, but upon termination (the use of $A \to a$) we start G_2 (by replacing the rule $A \to a$ by $A \to aS_2$).

The regular grammar generating L_1L_2 is given by $\langle\Sigma_1 \cup \Sigma_2,\ N_1 \cup N_2,\ P,\ S_1\rangle$, where P is defined by:

$$\begin{array}{rcl} P \stackrel{def}{=} & & \{A \to aB \mid A \to aB \in P_1\} \\ & \cup & \{A \to aS_2 \mid A \to a \in P_1\} \\ & \cup & P_2 \setminus \{S_2 \to \varepsilon\} \end{array}$$

This assumes that both languages are ε-free. If $S_1 \to \varepsilon \in P_1$, we add to the productions P, the set $\{S_1 \to \alpha \mid S_2 \to \alpha \in P_2\}$. If $S_2 \to \varepsilon \in P_2$, we add

$\{A \to a \mid A \to a \in P_1\}$ to P. If both have an initial ε-production, we add both sets.

Example: Consider the following two regular grammars:

$$\begin{array}{l} G_1 = \langle\{\mathtt{a}\}, \{S, A\}, P_1.S\rangle \\ \qquad\qquad P_1 \;=\; \{S \to \mathtt{a}A \mid \mathtt{a},\ A \to \mathtt{a}S \mid \mathtt{a}\} \\ G_2 = \langle\{\mathtt{b}\}, \{S, B\}, P_2.S\rangle \\ \qquad\qquad P_2 \;=\; \{S \to \mathtt{b}B \mid \mathtt{b},\ B \to \mathtt{b}S \mid \mathtt{b}\} \end{array}$$

Clearly, G_1 just generates strings composed of an arbitrary number of as whereas G_2 does the same but with bs. If we desire to define a regular grammar generating strings of as or bs (but not a mixture of both), we need a grammar G which satisfies:

$$\mathcal{L}(G) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$$

Using the last theorem we can obtain such a grammar mechanically.

First note that grammars G_1 and G_2 have a common non-terminal symbol S. To avoid problems, we rename the S in G_1 to S_1 (similarly in G_2).

We can now simply apply the method and define:

$$G = \langle\{\mathtt{a}, \mathtt{b}\}, \{S_1, S_2, A, B, S_\cup\}, P, S_\cup\rangle$$

P now contains all transitions in P_1 and P_2 (except empty ones which we do not have) and extra transitions from the new start symbol $S_\cup$ to strings α (where we have a rule $S_i \to \alpha$ in P_i):

$$\begin{array}{lcll} P & = & \{ & S_\cup \to \mathtt{a}A \mid \mathtt{b}B \mid \mathtt{a} \mid \mathtt{b} \\ & & & S_1 \to \mathtt{a}A \mid \mathtt{a}, \\ & & & S_2 \to \mathtt{b}B \mid \mathtt{b}, \\ & & & A \to \mathtt{a}A \mid \mathtt{a} \\ & & & B \to \mathtt{a}B \mid \mathtt{b} \qquad\qquad \} \end{array}$$

Theorem: If L_1 and L_2 are both regular languages, then so is their union $L_1 \cup L_2$.

Strategy: Again, we do not give a proof but just a construction.

Let L_i be generated by regular grammar $G_i = \langle \Sigma_i,\ N_i,\ P_i,\ S_i \rangle$. Again, we assume that the non-terminal symbols are disjoint, otherwise, we rename them. The idea is to start off from a new start symbol which may evolve like either S_1 or S_2. We do not remove the rules for S_1 and S_2, since they may appear on the right hand side.

The regular grammar generating $L_1 \cup L_2$ is given by $\langle \Sigma_1 \cup \Sigma_2,\ N_1 \cup N_2,\ P,\ S \rangle$, where P is defined by:

$$
\begin{array}{rcl}
P \stackrel{def}{=} & & \{S \to \alpha \mid S_1 \to \alpha \in P_1\} \\
& \cup & \{S \to \alpha \mid S_2 \to \alpha \in P_2\} \\
& \cup & P_1 \\
& \cup & P_2
\end{array}
$$

Example: These construction methods allow us to calculate grammars for complex languages from simpler ones, hence reducing or doing away with certain proofs altogether. This is what this example tries to demonstrate.

Suppose that we need a regular grammar which recognizes exactly those strings built up from sequences of double letters, over the alphabet $\{\mathtt{a}, \mathtt{b}\}$. Hence, `aabbaaaa` is acceptable, whereas `aabbbaa` is not. The language can be expressed as the set: $\{\mathtt{aa}, \mathtt{bb}\}^*$.

But it is trivial to prove that: $\{\mathtt{aa}, \mathtt{bb}\}$ is the same as $\{\mathtt{aa}\} \cup \{\mathtt{bb}\}$, where $\{\mathtt{aa}\} = \{\mathtt{a}\}\{\mathtt{a}\}$ (and similarly for $\{\mathtt{bb}\}$). We have thus decomposed our specification into:

$$(\{\mathtt{a}\}\{\mathtt{a}\} \cup \{\mathtt{b}\}\{\mathtt{b}\})^*$$

Note that all the operations (catenation, Kleene closure, union) are closed under the set of regular languages. The only remaining job is to obtain a regular grammar for the language $\{\mathtt{a}\}$ and $\{\mathtt{b}\}$, which is trivial.

Let $G_{\mathtt{a}}$ be the grammar producing $\{\mathtt{a}\}$.

$$G_{\mathtt{a}} = \langle \{\mathtt{a}\}, \{S\}, \{S \to \mathtt{a}.S \rangle$$

Similarly, we can define $G_{\mathtt{b}}$. The proof that $\mathcal{L}(G_{\mathtt{a}}) = \{\mathtt{a}\}$ is trivial.

Using the regular language catenation theorem, we can now construct a grammar to recognize $\{\texttt{a}\}\{\texttt{a}\}$:

$$G_{\texttt{aa}} = \langle \{\texttt{a}\}, \{S, A\}, \{S \to \texttt{a}A,\ A \to \texttt{a}\}, S\rangle$$

From $G_{\texttt{aa}}$ and $G_{\texttt{bb}}$ we can now construct a regular grammar which recognizes the union of the two languages.

$$G_\cup = \langle \{\texttt{a}, \texttt{b}\}, \{S, S_1, S_2.A, B\}, P_\cup.S\rangle$$

$$\begin{array}{rcll} P_\cup & = & \{ & S \to \texttt{a}A \mid \texttt{b}B, \\ & & & S_1 \to \texttt{a}A, \\ & & & S_2 \to \texttt{b}B, \\ & & & A \to \texttt{a}, \\ & & & B \to \texttt{b} \qquad \} \end{array}$$

Finally, from this we generate a grammar which recognizes the language $(\mathcal{L}(G_\cup))^*$. This is done in two steps: by first defining G^+, and then adding ε to the language:

$$G^+ = \langle \{\texttt{a}, \texttt{b}\}, \{S, S_1, S_2.A, B\}, P^+, S\rangle$$

$$\begin{array}{rcll} P^+ & = & \{ & S \to \texttt{a}A \mid \texttt{b}B, \\ & & & S_1 \to \texttt{a}A, \\ & & & S_2 \to \texttt{b}B, \\ & & & A \to \texttt{a} \mid \texttt{a}S, \\ & & & B \to \texttt{b} \mid \texttt{b}S \qquad \} \end{array}$$

$$G^* = \langle \{\texttt{a}, \texttt{b}\}, \{Z, S, S_1, S_2.A, B\}, P^*, Z\rangle$$

$$\begin{array}{rcll} P^* & = & \{ & Z \to \varepsilon \mid \texttt{a}A \mid \texttt{b}B, \\ & & & S \to \texttt{a}A \mid \texttt{b}B, \\ & & & S_1 \to \texttt{a}A, \\ & & & S_2 \to \texttt{b}B, \\ & & & A \to \texttt{a} \mid \texttt{a}S, \\ & & & B \to \texttt{b} \mid \texttt{b}S \qquad \} \end{array}$$

Note that by construction:

$$
\begin{array}{ll}
& \mathcal{L}(G^*) \\
= & (\mathcal{L}(G_\cup))^* \\
= & (\mathcal{L}(G_{\texttt{aa}}) \cup \mathcal{L}(G_{\texttt{bb}}))^* \\
= & (\mathcal{L}(G_{\texttt{a}})\mathcal{L}(G_{\texttt{a}}) \cup \mathcal{L}(G_{\texttt{b}})\mathcal{L}(G_{\texttt{b}}))^* \\
= & (\{\texttt{a}\}\{\texttt{a}\} \cup \{\texttt{b}\}\{\texttt{b}\})^* \\
= & (\{\texttt{aa}\} \cup \{\texttt{bb}\})^* \\
= & (\{\texttt{aa}, \texttt{bb}\})^*
\end{array}
$$

Exercises

Consider the following two regular grammars:

$$
\begin{array}{llll}
G_1 = \langle \{\texttt{a}, \texttt{b}, \texttt{c}\}, \{S, A\}, P_1, S\rangle & & & \\
& P_1 & = & \{ \quad S \to \texttt{a}S \mid \texttt{b}S \mid \texttt{a}A \mid \texttt{b}A, \\
& & & \quad\;\; A \to \texttt{c}A \mid \texttt{c} \quad \} \\
G_2 = \langle \{\texttt{a}, \texttt{b}, \texttt{c}\}, \{S, A\}, P_2.S\rangle & & & \\
& P_1 & = & \{ \quad S \to \texttt{c}S \mid \texttt{c}A, \\
& & & \quad\;\; A \to \texttt{a}A \mid \texttt{b}A \mid \texttt{a} \mid \texttt{b} \quad \}
\end{array}
$$

Let $L_1 = \mathcal{L}(G_1)$ and $L_2 = \mathcal{L}(G_2)$. Using the construction methods described in this chapter, construct a regular grammars to recognize the following languages:

1. $L_1 \cup L_2$
2. $L_1 L_2$
3. L_1^*
4. $L_1(L_2^*)$

Prove that $\forall w : \Sigma^* \cdot w \in L_1 \Leftrightarrow w^R \in L_2$.

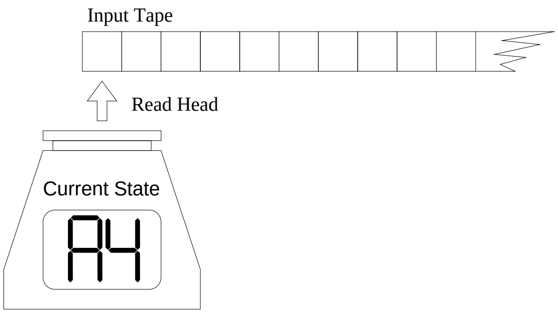
3.4 Conclusions

Regular languages and context free languages provide apparently sensible ways of classifying general phrase structure grammars. The motivations for choosing these subsets should become clearer in the chapters that follow. The next chapter proves some properties of these language classes. We then investigate a number of means of defining grammars as alternatives to using grammars. For each new method we relate the set of languages recognized with the classes we have defined.

Chapter 4

Finite State Automata

4.1 An Informal Introduction



Imagine a simple machine, an automaton, which can be in one of a finite number of states and can sequentially read input off a tape. Its behaviour is determined by a simple algorithm:

1. Start off from a pre-determined starting state and from the beginning of the tape.

2. Read in the next value off the tape.

3. Depending on the current state and the value just read determine the next state (via some internal table).

4. If the current state is a terminal one (from an internal list of terminal states) the machine *may* stop.

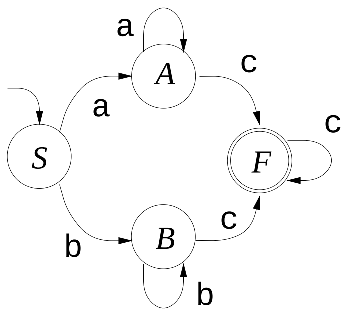
5. Advance the tape by one position.

6. Jump back to instruction 2.

But why are finite state automata and formal languages combined in one course? The short discussion in the introduction should be enough to answer this question: we can define the language accepted by a machine to be those strings which, when put on the input tape, may cause the automaton to terminate. Note that if an automaton reads a symbol for which it has no action to perform in the current state, it is assumed to 'break' and the string is rejected.

4.1.1 A Different Representation

In the introduction, automatons were drawn in a more abstract, less 'realistic' way. Consider the following diagram:



Every labeled circle is a *state* of the machine, where the label is the name of the state. If the internal transition table says that from a state A and with input `a`, the machine goes to state B, this is represented by an arrow labeled `a` going from the circle labeled A to the circle labeled B. The initial state from which the automaton originally starts is marked by an unlabeled incoming arrow. Final states are drawn using two concentric circles rather than just one.

Imagine that the machine shown in the previous diagram were to be given the string `aac`. The machine starts off in state S with input `a`. This sends the machine to state A, reading input `a`. Again, this sends the machine to state A, this time reading input `c`, which sends it to state F. The input string has finished, and the automaton has ended in a final state. This means that the string has been accepted.

Similarly, with input string `bcc` the automaton visits these states in order: S, B, F, F. Since after finishing with the string, the machine has ended in a terminal state, we conclude that `bcc` is also accepted.

Now consider the input `a`. Starting from S the machine goes to A, where the input string finishes. Since A is not a terminal state `a` is not an accepted string.

Alternatively consider any string starting with `c`. From state S, there is no outgoing arrow labeled `c`. The machine thus 'breaks' and thus `c` is not accepted.

Finally consider the string `aca`. The automaton goes from S (with input `a`) to A (with input `c`) to F (with input `a`). Here, the machine 'breaks' and the string is rejected. Note that even though the machine broke in a terminal state the string is not accepted.

4.1.2 Automata and Languages

Using this criterion to determine which strings are accepted and which are not, we can identify the set of strings accepted and call it the 'language' generated by the automaton. In other words, the behaviour of the automaton is comparable to that of a grammar, in that it identifies a set of strings.

The language intuitively accepted by the automaton depicted earlier is:

$$\{\mathtt{a}^n\mathtt{c}^m \mid n, m \geq 1\} \cup \{\mathtt{b}^n\mathtt{c}^m \mid n, m \geq 1\}$$

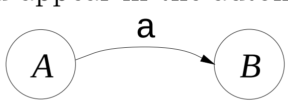
This (as yet informal) description for languages accepted by automata raises a number of questions which we will answer in this course.

- How can the concept of automata be formalized to allow rigorous proofs of their properties?
- What is the set of languages which can be accepted by automata. Is it as general (or even more general) than the set of languages which can be generated from phrase structure grammars, or is it more limited and can manage only context free or regular languages (or possibly even less)?

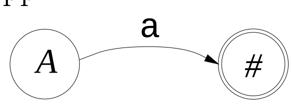
4.1.3 Automata and Regular Languages

Recall the definition of regular grammars. All productions were of the form $A \to a$ or $A \to aB$. Recall also that all sentential forms had exactly one non-terminal. What if we associate a state with every non terminal?

Every rule of the form $A \to aB$ means that non-terminal (state) A can evolve to B with input a. It would appear in the automaton as:



Rules of the form $A \to a$ mean that non-terminal (state) A can terminate with input a. This would appear in the automaton as:

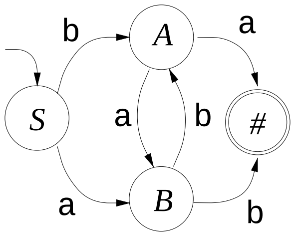


The starting state would simply be the initial non-terminal.

Example: Consider the regular grammar $G = \langle \Sigma,\ N,\ P,\ S \rangle$:

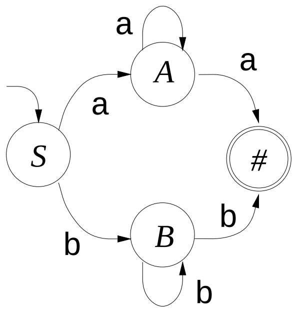
$$
\begin{array}{rcl}
\Sigma & = & \{\mathtt{a},\ \mathtt{b}\} \\
N & = & \{S,\ A,\ B\} \\
P & = & \{\ \ S \to \mathtt{a}B \mid \mathtt{b}A \\
 & & \quad\ A \to \mathtt{a}B \mid \mathtt{a} \\
 & & \quad\ B \to \mathtt{b}A \mid \mathtt{b} \quad \}
\end{array}
$$

Using the recipe just given to construct the finite state automaton from the regular grammar G, we get:



Note that states A and B exhibit a new type of behaviour: non-determinism. In other words, if the automaton is in state A and is given input a, its next state is not predictable. It can either go to B or to $\#$. In such automatons, a string would be accepted if there is at least one path of execution which ends in a terminal state.

Thus, intuitively at least, it seems that every regular grammar has a corresponding automaton with the same language, making automatons at least as expressive as regular grammars. Are they more expressive than that? The next couple of examples address this problem.



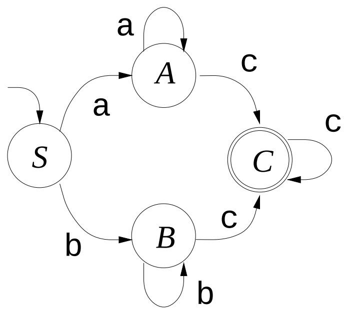
Consider the finite state automaton above. Let us define a grammar such that the non-terminal symbols are the states.

Thus, we have a grammar $G = \langle \Sigma,\ N,\ P,\ S \rangle$, where $\Sigma = \{\mathtt{a},\ \mathtt{b}\}$ and $N = \{S,\ A,\ B,\ \#\}$. What about the production rules?

Clearly, the productions $S \rightarrow \mathtt{a}A \mid \mathtt{b}B$ are in P. Similarly, so are $A \rightarrow \mathtt{a}A$ and $B \rightarrow \mathtt{b}B$. What about the rest of the transitions? Clearly, once from A the machine goes to $\#$ it has no alternative but to terminate. Thus we add $A \rightarrow \mathtt{a}$ and similarly for B, $B \rightarrow \mathtt{b}$. The resulting grammar is thus:

$$
\begin{array}{rcl}
\Sigma & = & \{\mathtt{a},\ \mathtt{b}\} \\
N & = & \{S,\ A,\ B,\ \#\} \\
P & = & \{\ \ S \rightarrow \mathtt{a}A \mid \mathtt{b}B \\
 & & \quad\ \ A \rightarrow \mathtt{a}A \mid \mathtt{a} \\
 & & \quad\ \ B \rightarrow \mathtt{b}B \mid \mathtt{b} \quad \}
\end{array}
$$

However, sometimes things may not be so clear. Consider:



Using the same strategy as before, we construct grammar $G = \langle \Sigma,\ N,\ P,\ S \rangle$, where $\Sigma = \{\mathtt{a},\ \mathtt{b},\ \mathtt{c}\}$ and $N = \{S,\ A,\ B,\ C\}$. As for the production rules, some are rather obvious to construct:

$$\begin{array}{rcl} S & \rightarrow & \mathtt{a}A \mid \mathtt{b}B \\ A & \rightarrow & \mathtt{a}A \\ B & \rightarrow & \mathtt{b}B \end{array}$$

What about the rest? Using the same design principle as for the rules just given, we also give:

$$\begin{array}{rcl} A & \rightarrow & \mathtt{c}C \\ B & \rightarrow & \mathtt{c}C \\ C & \rightarrow & \mathtt{c}C \end{array}$$

However, C may also terminate without requiring any further input. This corresponds to $C \rightarrow \varepsilon$, where C stops getting any further input and terminates (no more non-terminals).

Note that all the production rules we construct, except for the ε-productions, are according to the restrictions placed for regular grammars. However, we have already identified a method for generating a regular grammar in cases when we have ε-productions:

1. Notice that in P, ε is derivable from the non-terminal C only.

2. We now construct the grammar G' where we copy all rules in P but where we also copy the rules which use C without it:

$$\begin{array}{rcl} S & \rightarrow & \mathtt{a}A \mid \mathtt{b}B \\ A & \rightarrow & \mathtt{a}A \mid \mathtt{c}C \mid \mathtt{c} \\ B & \rightarrow & \mathtt{b}B \mid \mathtt{c}C \mid \mathtt{c} \\ C & \rightarrow & \mathtt{c}C \mid \mathtt{c} \end{array}$$

3. The grammar is now regular. (Note that in some cases it may have been necessary modify this grammar, had we ended with one in which we have the rule $S \rightarrow \varepsilon$ and S also appears on the right hand side of some rules)

Thus it seems possible to associate a regular grammar with every automaton. This informal reasoning thus suggests that automata are exactly as expressive as regular grammars. That is what we now set out to prove.

4.2 Deterministic Finite State Automata

We start by formalizing deterministic finite state automata. This is the class of all automata as already informally discussed, except that we do not allow non-deterministic edges. In other words, there may not be more that one edge leaving any state with the same label.

Definition: A deterministic finite state automaton M, is a 5-tuple:

M	$= \langle K,\ T,\ t,\ k_1,\ F \rangle$
K	is a finite set of states
T	is the finite input alphabet
t	is the partial transition function which, given a state and an input, determines the new state: $K \times T \rightarrow K$
k_1	is the initial state ($k_1 \in K$)
F	is the set of final (terminal) states ($F \subseteq K$)

Note that t is partial, in that it is not defined for certain state, input combinations.

Definition: Given a transition function t (type $K \times T \to K$), we define its string closure t_* (type $K \times T^* \to K$) as follows:

$$\begin{aligned} t_*(k,\ \varepsilon) &\stackrel{def}{=} k \\ t_*(k,\ as) &\stackrel{def}{=} t_*(t(k,a),\ s) \text{ where } a \in T \text{ and } s \in T^* \end{aligned}$$

Intuitively, $t_*(k,\ s)$ returns the last state reached upon starting the machine from state k with input s.

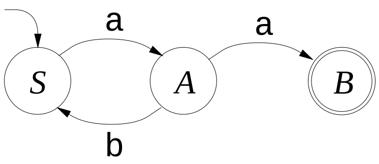
Proposition: $t_*(A,\ xy) = t_*(t_*(A,x),\ y)$ where $x, y \in T^*$.

Definition: The set of strings (language) accepted by the deterministic finite state automaton M is denoted by $\mathcal{T}(M)$ and is the set of terminal strings x, which, when M is started from state k_1 with input x, it finishes in one of the final states. Formally it is defined by:

$$\mathcal{T}(M) \stackrel{def}{=} \{x \mid x \in T^*,\ t_*(k_1,\ x) \in F\}$$

Definition: Two deterministic finite state automata M_1 and M_2 are said to be equivalent if they accept the same language: $\mathcal{T}(M_1) = \mathcal{T}(M_2)$.

Example: Consider the automaton depicted below:



Formally, this is $M = \langle K,\ T,\ t,\ S,\ F\rangle$ where:

$$\begin{aligned} K &= \{S,\ A,\ B\} \\ T &= \{a,\ b\} \\ t &= \{(S,a) \to A,\ (A,a) \to B,\ (A,b) \to S\} \\ F &= \{B\} \end{aligned}$$

We now prove that every string of the form $a(ba)^n a$ (where $n \geq 0$) is generated by M.

We require to prove that $\{a(ba)^n a \mid n \geq 0\} \subseteq \mathcal{T}(M)$

We first prove that $t_*(S, a(ba)^n) = A$ by induction on n.

Base case ($n = 0$): $t_*(S, a) = t(S, a) = A$.

Inductive case: Assume that $t_*(S, a(ba)^k) = A$.

$$\begin{array}{lll} & t_*(S, a(ba)^{k+1}) & \\ = & t_*(S, a(ba)^k ba) & \\ = & t_*(t_*(S, a(ba)^k), ba) & \text{by property of } t_* \\ = & t_*(A, ba) & \text{by inductive hypothesis} \\ = & A & \text{by applying } t_* \end{array}$$

Completing the induction. Hence, for any n:

$t_*(S, a(ba)^n a) = t(t_*(S, a(ba)^n), a) = t(A, a) = B \in F$

Hence $a(ba)^n a \in \mathcal{T}(M)$.

Proposition: For any deterministic finite state automaton M, we can construct an equivalent finite state automaton M' with a total transition function.

Strategy: We add a new dummy state Δ (and is not a final state) to which all previously 'missing' arrows point.

Proof: Define deterministic finite state machine M' as follows: $M' = \langle K \cup \{\Delta\},\ T,\ t',\ k_1,\ F\rangle$ where $t'(k, x) = t(k, x)$ whenever $t(k, x)$ is defined, $t'(k, x) = \Delta$ otherwise.

Part 1: $\mathcal{T}(M) \subseteq \mathcal{T}(M')$

Consider $x \in \mathcal{T}(M)$. This implies that $t_*(k_1,\ x) \in F$. But by definition, whenever t_* is defined, so is t'_*. Furthermore they give the same value, and thus $t'_*(k_1,\ x) = t_*(k_1,\ x) \in F$. Hence $x \in \mathcal{T}(M')$.

Part 2: $\mathcal{T}(M') \subseteq \mathcal{T}(M)$

We start by proving, by induction on the length of x that, if $t'_*(A,\ x) \in F$ then $t_*(A,\ x) \in F$.

Base case ($|x| = 0$): Implies that $A \in F$ and $x = \varepsilon$. Hence $t_*(A,\ x) \in F$.

Inductive case: Assume that the argument holds for $|x| = k$. Now consider $x = ay$, where $|x| = k + 1$ and $a \in T$.

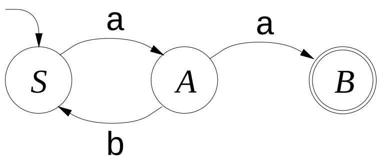
Notice that $t'_*(A,\ ay) = t'_*(t'(A,a),\ y)$. Hence, by induction hypothesis $t_*(t'(A,a),\ y) \in A$. But $t'(A,a) \neq \Delta$ (otherwise t_* would not be defined on it), and therefore $t(A,a) = t'(A,a)$.

Therefore, $t_*(t(A,a),\ y) \in A$, implying that $t_*(A,\ ay) \in A$ completing the induction.

Therefore, by this result $t'_*(k_1,\ x) \in F$ implies that $t_*(k_1,\ x) \in F$. Hence $x \in \mathcal{T}(M')$ implies that $x \in \mathcal{T}(M)$, completing the proof.

□

Example: Consider the automaton given in the previous example.

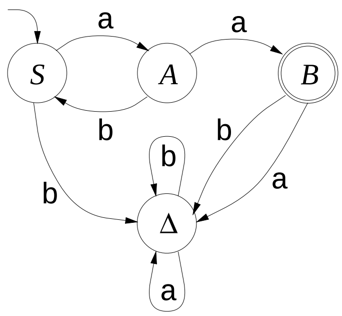


We can define a automaton M' with a total transition function equivalent to M by using the method just prescribed in the previous theorem.

Formally, this would be $M' = \langle K',\ T,\ t',\ S,\ F\rangle$ where:

$$
\begin{array}{rcl}
K' & = & \{S,\ A,\ B,\ \Delta\} \\
T & = & \{a,\ b\} \\
t' & = & \{\ \ (S,a) \to A, \\
 & & \quad\ (S,b) \to \Delta, \\
 & & \quad\ (A,a) \to B, \\
 & & \quad\ (A,b) \to S, \\
 & & \quad\ (B,a) \to \Delta, \\
 & & \quad\ (B,b) \to \Delta, \\
 & & \quad\ (\Delta,a) \to \Delta, \\
 & & \quad\ (\Delta,b) \to \Delta \quad \} \\
F & = & \{B\}
\end{array}
$$

This can be visually represented by:



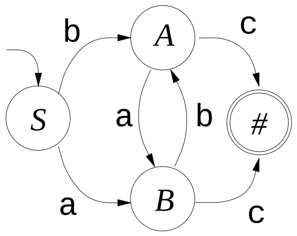
4.2.1 Implementing a DFSA

Given the transition function of a finite state automaton, we can draw up a table of state against input symbol and fill in the next state.

	a	a	...
A	$t(A, \mathtt{a})$	$t(B, \mathtt{a})$	...
B	$t(A, \mathtt{b})$	$t(B, \mathtt{b})$	...
⋮	⋮	⋮	⋱

Combinations of state against input for which the transition function is undefined, are left empty.

Example: Consider the following automaton:



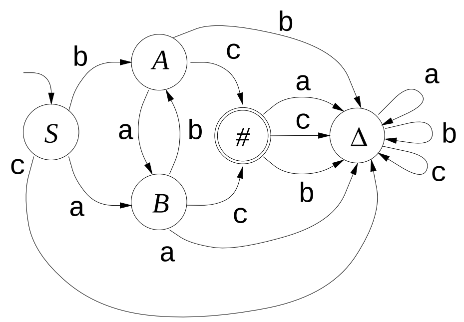
The transition function of this automaton is:

	a	b	c
S	B	A	
A	B		$\#$
B		A	$\#$
$\#$			

Adding a dummy state Δ to make the transition function total is very straightforward:

	a	b	c
S	B	A	Δ
A	B	Δ	$\#$
B	Δ	A	$\#$
$\#$	Δ	Δ	Δ
Δ	Δ	Δ	Δ

Visually, the automaton with a total transition function would look like:



This representation of the transition function shows how a deterministic finite state automaton can be implemented. A 2-dimensional array is used to model the transition function.

4.2.2 Exercises

1. Write a program in the programming language of your choice which implements a deterministic finite state automaton. The automaton parameters (such as the transition function) need not be inputted by the user but can be initialized from within the program itself.

   Most importantly, provide a function which takes an input string and returns `true` or `false` depending on whether the the string is accepted by the automaton.

2. Give a necessary and sufficient condition for the language generated by a DFSA to include ε.

3. Prove that, if a non-terminal state has no outgoing transitions, then if we remove that state (and transitions going into it), the resultant automaton is equivalent to the original one.

4. Prove that any state (except the starting state) which has no incoming transitions can be safely removed from the automaton without affecting the language accepted.

4.3 Non-deterministic Finite State Automata

Recall that in the initial introduction to automata we sometimes had multiple transitions from the same state and with the same label. These are not deterministic finite state automata since we had defined a transition *function*, and thus we can have only one value for the application $t(A, \mathtt{a})$. How can we generalize the concept of automata to allow for non-determinism?

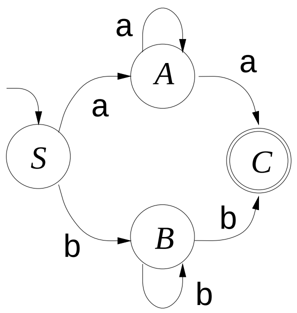
Definition: A non-deterministic finite state automaton M is a 5-tuple $\langle K,\ T,\ t,\ k_1,\ F\rangle$, where:

K	is a finite set of states in which the automaton can reside
T	is the input alphabet
$t_1 \in K$	is the initial state
$F \subseteq K$	is the set of final states

t is a total function from state and input to a set of states. $(K \times T \to \mathbb{P}K)$.

The set of states $t(A, \mathtt{a})$ defines all possible new states if the machine reads an input $\mathtt{a}$ in state A. If no transition is possible, for such a state, input combination $t(A, \mathtt{a}) = \emptyset$.

Example: Consider the non-deterministic automaton depicted below:



This would formally be encoded as non-deterministic finite state automaton $M = \langle \{S, A, B, C\},\ \{\mathtt{a}, \mathtt{b}\},\ t,\ S,\ \{C\}\rangle$, where t is:

$$
\begin{array}{rcll}
t & = & \{ & (S,\mathtt{a}) \to \{A\}, \\
 & & & (S,\mathtt{b}) \to \{B\}, \\
 & & & (A,\mathtt{a}) \to \{A,\ C\}, \\
 & & & (A,\mathtt{b}) \to \emptyset, \\
 & & & (B,\mathtt{a}) \to \emptyset, \\
 & & & (B,\mathtt{b}) \to \{B,\ C\}, \\
 & & & (C,\mathtt{a}) \to \emptyset, \\
 & & & (C,\mathtt{b}) \to \emptyset \qquad\qquad \}
\end{array}
$$

Definition: The extension of a transition function to take sets of states $\bar{t} : \mathbb{P}K \times T \to \mathbb{P}K$, is defined by:

$$\bar{t}(S,\mathtt{a}) \quad \stackrel{def}{=} \quad \bigcup_{k\in S} t(k,\mathtt{a})$$

Example: In the previous example, imagine the machine has received input $\mathtt{a}$ in state A. Clearly, the new state is either A or C. Where would the machine now reside if we received another input $\mathtt{a}$?

$$
\begin{array}{rl}
 & \bar{t}(\{A,C\},\mathtt{a}) \\
= & t(A,\mathtt{a}) \cup t(C,\mathtt{a}) \\
= & \{A,C\} \cup \emptyset \\
= & \{A,C\}
\end{array}
$$

Definition: As before, we can now define t_*, the string closure of $\bar{t}$:

$$
\begin{array}{rcl}
t_*(S,\ \varepsilon) & \stackrel{def}{=} & S \\
t_*(S,\ as) & \stackrel{def}{=} & t_*(\bar{t}(S,a),\ s) \text{ where } a \in T \text{ and } s \in T^*
\end{array}
$$

Definition: A non-deterministic finite state automaton M is said to accept a terminal string s, if, starting from the initial state with input s, it may reach a final state: $t_*(k_1,\ s) \cap F \neq \emptyset$.

Definition: The language accepted by a non-deterministic finite state automaton M, is the set of all strings (in T^*) accepted by M:

$$\mathcal{T}(M) \stackrel{def}{=} \{s \mid s \in T^*,\ t_*(k_1,\ s) \cap F \neq \emptyset\}$$

How does the class of languages accepted by non-deterministic finite state automata compare with those accepted by the deterministic variety, and with those generated by context-free or regular grammars? Thus is what we now set out to discuss.

4.4 Formal Comparison of Language Classes

Implementation of a non-deterministic finite state automaton using a computer language is not as simple as writing one for a deterministic automaton. The cells in the transition table would have to hold lists of states, rather than just one state. Checking whether a string is accepted or not is also not as simple, since a 'wrong' transition early on could mean that a terminal state is not reached and to make sure that a string is not accepted would mean having to search through all the possibilities exhaustively.

This seems to indicate that non-determinism adds complexity. Does this mean, however, that there are languages which a non-deterministic finite state automaton can accept but for which there is no deterministic finite state automaton which accepts them? The next two theorems prove that this is not the case, and that (surprise, surprise!) for every non-deterministic finite state automaton, there is a deterministic finite state automaton which accepts the same language (and vice-versa).

Theorem: Every language recognizable by a deterministic finite state automaton is also recognizable by a non-deterministic finite state automaton.

Proof: Given a DFSA $M = \langle K,\ T,\ t,\ k_1,\ F\rangle$, build an equivalent DFSA $M' = \langle K',\ T',\ t',\ k'_1,\ F'\rangle$, but in which the transition function is total.

We now define a NFSA $M'' = \langle K',\ T',\ t'',\ k'_1,\ F'\rangle$, where $t'' = \{(A, a) \to \{B\} \mid (A, a) \to B \in t'\}$.

We claim that the languages generated by M' and M'' are, in fact, identical. This can be proved by showing that the range of t''_* is in fact singleton sets and that $t''_*(A,a) = \{B\} \Leftrightarrow t'_*(A,a) = B$. This is easily provable by induction and will not be done here.

Now, consider $x \in \mathcal{T}(M')$. This is implies that $t'_*(A,x) = B$ and $B \in F$. Hence, by the above reasoning, $t''_*(A,x) = \{B\}$ and $\{B\} \in F$, which in turn implies that $t''_*(A,x) \cap F \neq \emptyset$. Thus, $x \in \mathcal{T}(M'')$.

Conversely, if $x \in \mathcal{T}(M'')$, $t''_*(A,x) \cap F \neq \emptyset$. But $t''_*(A,x) = \{B\}$, which implies that $\{B\} \cap F \neq \emptyset$, that is $B \in F$. But $t''_*(A,x) = \{B\}$ also implies (by the above reasoning) that $t'_*(A,x) = B$. Hence $t'_*(A,x) \in F$. $x \in \mathcal{T}(M')$.

Thus, $\mathcal{T}(M') = \mathcal{T}(M'')$.

□

Theorem: Conversely, if a language is recognizable by a non-deterministic finite state automaton, there is a deterministic finite state automaton which recognizes the language.

Strategy: The idea is to create a new automaton with every combination of states in the original represented in the new automaton by a state. Thus, if the original automaton had states A and B, we now create an automaton with 4 states labeled $\emptyset$, $\{A\}$, $\{B\}$ and $\{A,B\}$. The transitions would then correspond directly to $\bar{t}$. Thus, if $\bar{t}(\{A\},a) = \{A,B\}$ in the original automaton, we would have a transition labeled a from state $\{A\}$ to state $\{A,B\}$. The equivalence of the languages is then natural.

Proof: Given a DFSA $M = \langle K,\ T,\ t,\ k_1,\ F\rangle$, we now define a NFSA $M'' = \langle K',\ T,\ t',\ k'_1,\ F'\rangle$, where :

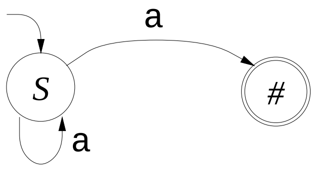
$$\begin{aligned} K' &= \mathbb{P}K \\ t' &= \bar{t} \\ k'_1 &= \{k_1\} \\ F' &= \{S \mid S \subseteq K,\ S \cap F \neq \emptyset\} \end{aligned}$$

$$\begin{aligned} & x \in \mathcal{T}(M) \\ \Leftrightarrow\ & \text{by definition of } \mathcal{T}(M) \\ & \bar{t}_*(\{k_1\},x) \cap F \neq \emptyset \end{aligned}$$

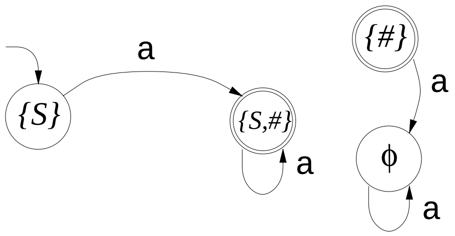
$$\begin{array}{ll} \Leftrightarrow & \text{by definition of } F' \\ & \bar{t}_*(\{k_1\}, x) \in F' \\ \Leftrightarrow & \text{by definition of } \mathcal{T}(M') \\ & x \in \mathcal{T}(M') \end{array}$$

□

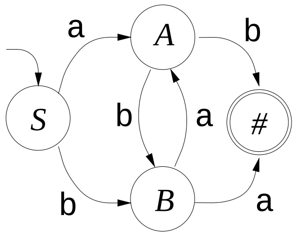
Example: Consider the non-deterministic finite state automaton below:



To produce a deterministic finite state automaton with the same language, we need a state for every combination of original states. Thus, we will now have the states $\emptyset$, $\{S\}$, $\{\#\}$ and $\{S, \#\}$. The starting state is $\{S\}$, whereas the final states are all those which include # ($\{\#\}$ and $\{S, \#\}$). By constructing $\bar{t}$, we can now construct the desired automaton:



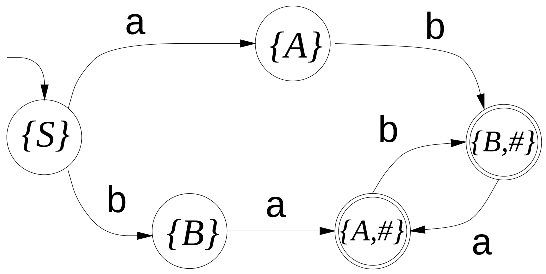
Example: For a slightly more complicated example, consider the non-deterministic finite state automaton below:



Since, for any finite set of size n, its power set is of size 2^n, we will now have 2^4 or 16 states!

To simplify the presentation we constrain the automaton by leaving out states which are unreachable from the starting state ($\{S, A\}$, $\{S, B, A\}$ etc), and states from which no terminal state is reachable ($\emptyset$).

You can check out for yourselves that the deterministic automaton below is defined according to the rules set in the last theorem:



This method provides a simple way of implementing a non-deterministic finite state automaton ... by first obtaining an equivalent deterministic one. As the last example indicates, the number of states in the deterministic grows pretty quickly even for small numbers of states in the original. A transformed 10 state non-deterministic automaton would have over 1000 states. A 20 state one would end up with more that 1,000,000 states! This indicates the importance of methods to reduce the size of automatons by removing irrelevant states. Some simple results were presented in the exercises of section 4.2.2, but this is beyond the scope of this course and will not be discussed any further here.

We still have not answered the question of how the languages generated by these automata relate to context-free and regular automata. The next theorem should answer this.

Theorem: Non-deterministic finite state automata, deterministic finite state automata and regular grammars generate exactly the same class of languages. The following statements are equivalent:

1. L is a regular language
2. L is accepted by a non-deterministic finite state automaton
3. L is accepted by a deterministic finite state automaton

Strategy: (2) and (3) have already been proved equivalent. In the introductory part of this chapter we presented a strategy for obtaining a (non-deterministic) automaton from a regular grammar and vice-versa. These are the methods which will be formalized here.

Proof: By the previous two theorems, we know that (2)⇔(3). We now show how that (3)⇒(1) and that (1)⇒(2). This will complete the proof thanks to transitivity of implication.

Part 1: (3)⇒(1). Let L be the language generated by deterministic finite state automaton $M = \langle K,\ T,\ t,\ k_1,\ F\rangle$. Now consider the regular grammar $G = \langle T,\ K,\ P,\ k_1\rangle$, where P is defined by:

$$\begin{aligned} P \stackrel{def}{=} & \quad \{A \to aB \mid t(A,a) = B\} \\ & \cup \ \{A \to a \mid t(A,a) \in F\} \end{aligned}$$

Claim: This generates $L \setminus \{\varepsilon\}$.

This can be proved by induction on the length of the derivation on the statements:

$$\begin{aligned} k_1 \stackrel{+}{\Rightarrow}_G aA \Leftrightarrow t_*(k_1, a) = A \\ k_1 \stackrel{+}{\Rightarrow}_G a \Leftrightarrow t_*(k_1, a) \in F \end{aligned}$$

If $\varepsilon \in L$ we can add this to the regular language using techniques already discussed.

Part 2: (1)⇒(2). Given grammar $G = \langle \Sigma,\ N,\ P,\ S\rangle$, we define the non-deterministic automaton $M = \langle N \cup \{\#\},\ \Sigma,\ t,\ S,\ \{\#\}\rangle$, where t is defined by:

$$\begin{aligned} t(A,a) \stackrel{def}{=} & \quad \{B \mid A \to aB \in P\} \\ & \cup \ \{\# \mid A \to a \in P\} \end{aligned}$$

This produces $\mathcal{L}(M) \setminus \{\varepsilon\}$.

If ε is in L, we need to add ε to the language accepted by M. This is done by defining $M' = \langle N \cup \{\#, k_1\},\ \Sigma,\ t',\ k_1,\ \{\#, k_1\}\rangle$, where t' is defined as:

$$\begin{aligned} t'(k_1, a) &\stackrel{def}{=} t(S, a) \\ t'(A, a) &\stackrel{def}{=} t(A, a) \text{ if } A \neq k_1 \end{aligned}$$

In other words, we add a new initial (and final) state k_1. This obviously adds the desired ε. How about the other inputs? State k_1 will also have outgoing transitions like S. This avoids problems with any arrows going back into S.

□

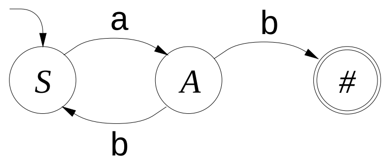
These equivalences will not be proved during the course. Our aim is mainly to show the important results in this particular area of computer science and the application of basic techniques. These proofs should be within your grasp. If you are interested in the details, the textbooks should provide adequate information.

This is one of the major results of the course. Most importantly, you should be able to transform between the three classes, including the special cases.

Example: Consider the grammar $\langle\{a, b\},\ \{S,\ A\},\ P,\ S\rangle$, where:

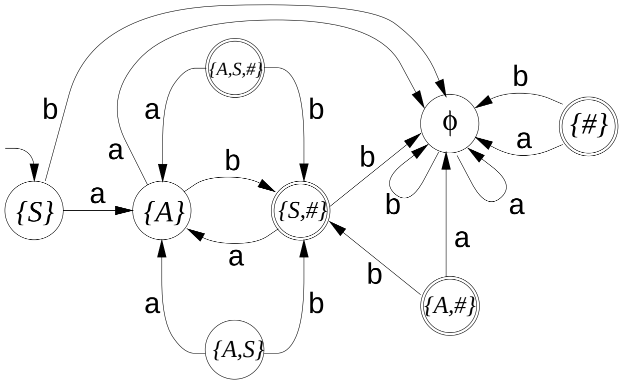
$$P \quad = \quad \{S \to aA,\ A \to b \mid bS\}$$

Using the method just given, we derive the non-deterministic finite state automaton recognizing the same language:



Since ε is not in the language, we need not perform any further modifications.

From this non-deterministic finite state automaton, we can construct a deterministic one:

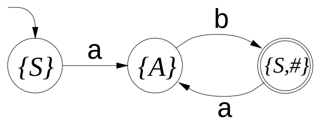


Finally, we produce a regular grammar from this automaton. If we rename states X by N_X (to avoid confusion) we get:

$$
\begin{array}{ll}
\langle & \{a,b\},\ \{N_\emptyset, N_{\{S\}}, \ldots N_{\{A,S,\#\}}\}, \\
& \quad \{ \quad N_{\{S\}} \rightarrow aN_{\{A\}} \mid bN_\emptyset \\
& \qquad\quad N_{\{A\}} \rightarrow b \mid bN_{\{S,\#\}} \mid aN_\emptyset \\
& \qquad\quad N_{\{A,S\}} \rightarrow b \mid aN_{\{A\}} \mid bN_{\{S,\#\}} \\
& \qquad\quad N_{\{A,\#\}} \rightarrow b \mid bN_{\{S,\#\}} \mid aN_\emptyset \\
& \qquad\quad N_{\{S,\#\}} \rightarrow aN_{\{A\}} \mid bN_\emptyset \\
& \qquad\quad N_{\{A,S,\#\}} \rightarrow b \mid aN_{\{A\}} \mid bN_{\{S,\#\}} \\
& \qquad\quad N_\emptyset \rightarrow aN_\emptyset \mid bN_\emptyset \\
& \qquad\quad N_{\{\#\}} \rightarrow aN_\emptyset \mid bN_\emptyset \qquad\qquad\qquad \} \\
& N_{\{S\}} \rangle
\end{array}
$$

Applying the methods used in the theorems, we are guaranteed that the initial and final grammars are equivalent.

Note that, to make the resultant grammar smaller, we can remove redundant states at the DFSA stage or unused non-terminals in the regular grammar stage. If we choose to make the DFSA more efficient, we note that the (non-initial) states $\{A,S,\#\}$, $\{A,S\}$, $\{A,\#\}$ and $\{\#\}$ have no in-coming transitions and can thus be left out. $\emptyset$ cannot evolve to any terminal state (for any number of moves) and can thus also be left out. The resultant DFSA would now look like:



The resulting smaller grammar would then be:

$$
\begin{array}{l}
\langle \quad \{a,b\},\ \{N_{\{S\}}, N_{\{A\}}, N_{\{S,\#\}}\}, \\
\quad \{ \quad N_{\{S\}} \rightarrow aN_{\{A\}} \\
\qquad\quad N_{\{A\}} \rightarrow b \mid bN_{\{S,\#\}} \\
\qquad\quad N_{\{S,\#\}} \rightarrow aN_{\{A\}} \qquad \} \\
\quad N_{\{S\}} \rangle
\end{array}
$$

These results are very useful in the implementation of regular language parsers. Given a regular language (or rather grammar), we have a safe, fail-proof method of designing a deterministic finite state automaton to recognize the language. Since it is particularly easy to program DFSA and check whether or not they accept a particular string, we can easily implement them. But the use of these theorems is not limited to implementation issues. They can help resolve questions about grammars by using arguments about finite state machines (or vice-versa).

Proposition: If L is a regular language over Σ, then so is its inverse $\overline{L}$ (where $\overline{L} = \Sigma^* \setminus L$).

Strategy: Since L is a regular grammar, there is a deterministic finite state automaton M which recognizes it. We now design an automaton $\overline{M}$ identical to M except that the a state S is a final state of $\overline{M}$ if and only if it is not a final state of M. The language recognized by $\overline{M}$ is exactly $\overline{L}$. Hence there is a regular grammar recognizing $\overline{L}$.

Proof: Since L is a regular language, there is a regular grammar generating it. By the theorem above, therefore, there is a DFSA recognizing it. Let $M = \langle \Sigma, T, t, k_1, F \rangle$ be such a DFSA. Assume that t is total (otherwise add a dummy state).

Consider $\overline{M} = \langle \Sigma, T, t, k_1, \Sigma \setminus F \rangle$. We claim that $\mathcal{T}(\overline{M}) = \Sigma^* \setminus \mathcal{T}(M)$.

$$x \in \mathcal{T}(\overline{M})$$

$$
\begin{aligned}
&\Rightarrow\ t_*(k_1, x) \in \Sigma \setminus F \\
&\Rightarrow\ t_*(k_1, x) \notin F \\
&\Rightarrow\ x \notin \mathcal{T}(\overline{M}) \\
&\Rightarrow\ x \in \Sigma \setminus \mathcal{T}(\overline{M})
\end{aligned}
$$

The reverse argument is very similar and is left out.

But $\Sigma^* \setminus \mathcal{T}(M) = \Sigma^* \setminus L = \overline{L}$ and hence $\mathcal{T}(\overline{M}) = \overline{L}$. Therefore, there is a DFSA generating $\overline{L}$, which, using the theorem means that there is a regular grammar generating it. Hence, $\overline{L}$ is a regular language.

□

Corollary: If L_1 and L_2 are regular languages, then so is $L_1 \cap L_2$.

Proof: $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$

□

Given these results, given two regular grammars, we can construct a regular grammar recognizing their intersection. Just using the results we gave in these notes, the procedure is rather lengthy but is easily automated:

1. Produce an equivalent NFSA for each of the two grammars (call them N_1 and N_2).
2. From the NFSA (N_1 and N_2) produce equivalent DFSA (call them D_1 and D_2).
3. Produce total DFSA ($\overline{D}_1$ and $\overline{D}_2$) recognizing the inverse of the languages recognized by D_1 and D_2.
4. From $\overline{D}_1$ and $\overline{D}_2$ produce equivalent regular grammars G_1 and G_2.
5. Construct a regular grammar $G_\cup$ recognizing the union of the grammars G_1 and G_2.
6. Produce a NFSA $N_\cup$ equivalent to $G_\cup$.
7. Produce a total DFSA $D_\cup$ equivalent to $N_\cup$.
8. From $D_\cup$ produce $\overline{D}_\cup$ which recognizes the inverse of the language.

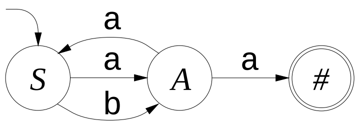
9. Construct a regular grammar G from DFSA $\overline{D}_\cup$.

The process may be lengthy and boring but it is guaranteed to work. In most cases, this process will produce an unnecessarily large grammar. However, as already mentioned, efficiency is not an issue which we will be looking into.

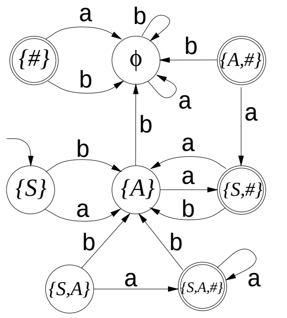
Example: Construct a grammar to recognize the inverse of the language generated by:

$$\langle \{\mathtt{a}, \mathtt{b}\}, \{S, A\}, \{S \to \mathtt{a}A \mid \mathtt{b}A, A \to \mathtt{a} \mid \mathtt{a}S\}, S\rangle$$

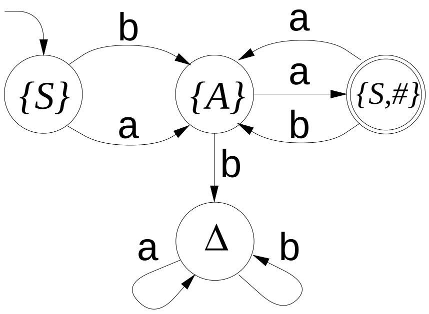
The non-deterministic finite state automaton we construct is:



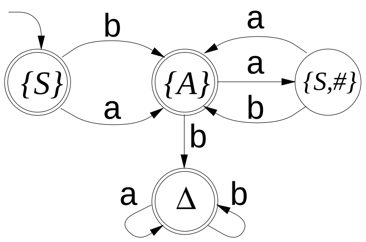
Note that ε is not in the language we derive and this NFSA does not require any modification. From this we generate the 8-state DFSA:



This is optimized and made total to:



From which we generate its inverse:



We can now generate a regular grammar $G = \langle \Sigma, N, P, \{S\} \rangle$.

$$\begin{array}{rcl} \Sigma & = & \{a, b\} \\ N & = & \{\{S\}, \{A\}, \{S, \#\}, \Delta\} \\ P & = & \{ \quad \{S\} \to \mathtt{a} \mid \mathtt{b} \mid \mathtt{a}\{A\} \mid \mathtt{b}\{A\} \\ & & \quad\quad \{A\} \to \mathtt{b} \mid \mathtt{a}\{S, \#\} \mid \mathtt{b}\Delta \\ & & \quad\quad \Delta \to \mathtt{a} \mid \mathtt{b} \mid \mathtt{a}\Delta \mid \mathtt{b}\Delta \\ & & \quad\quad \{S, \#\} \to \mathtt{a} \mid \mathtt{b} \mid \mathtt{a}\{A\} \mid \mathtt{b}\{A\} \quad \} \end{array}$$

Finally, since ε is an accepted string in the DFSA we derived the regular grammar from, we need to add ε to grammar G. Using the normal technique, we get $G' = \langle \Sigma, N \cup \{B\}, P', B \rangle$, where $P' = P \cup \{B \to \varepsilon \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{a}\{A\} \mid \mathtt{b}\{A\}\}$

4.4.1 Exercises

1. Calculate a DFSA which accepts strings in the regular language $G = \langle \Sigma, N, P, S \rangle$, where:

$$\begin{array}{rcl} \Sigma & = & \{\mathtt{a}, \mathtt{b}\} \\ N & = & \{S, A, B\} \\ P & = & \{ \quad S \to \mathtt{a}A \mid \mathtt{b}B \\ & & \quad\quad A \to \mathtt{b}S \mid \mathtt{b} \\ & & \quad\quad B \to \mathtt{a}S \mid \mathtt{a} \quad \} \end{array}$$

2. Construct a grammar G' which accepts $\mathcal{L}(G) \cup \{\varepsilon\}$, where G is as defined in the previous question. From G' construct a NFSA which recognizes the same language.

3. Construct a DFSA which recognizes $\overline{\mathcal{L}(G)}$, where G is the regular grammar $G = \langle \Sigma, N, P, S \rangle$, where:

$$
\begin{array}{rcl}
\Sigma & = & \{\mathtt{a}, \mathtt{b}\} \\
N & = & \{S, A, B\} \\
P & = & \{ \quad S \rightarrow \mathtt{a}B \mid \mathtt{a}A \\
 & & \quad\;\; A \rightarrow \mathtt{b}S \mid \mathtt{b} \\
 & & \quad\;\; B \rightarrow \mathtt{a}A \mid \mathtt{a} \quad \}
\end{array}
$$

Chapter 5

Regular Expressions

This chapter discusses another way of defining a language — by using regular expressions. This notation is particularly adept at describing simple languages in a concise and readable way. It is used in certain text editors (for example, `vi`) to specify a search for a particular sequence of symbols in the text. A similar notation is also used to specify the acceptable syntax of command lines.

5.1 Definition of Regular Expressions

Definition: A regular expression over an alphabet Σ takes the form of one of the following:

- 0
- 1
- `a` (where `a`$\in \Sigma$)
- (e) (where e is a regular expression)
- e^* (where e is a regular expression)
- e^+ (where e is a regular expression)
- $e_1 + e_2$ (where both e_1 and e_2 are regular expressions)

- e_1e_2 (where both e_1 and e_2 are regular expressions)

Thus, for example, all the following are regular expressions over the alphabet $\{\mathtt{a},\ \mathtt{b}\}$:

- $(\mathtt{a}+\mathtt{b})^*$
- $\mathtt{a}^*\mathtt{b}^*$
- $(\mathtt{ab}+\mathtt{ba})^+$
- $\mathtt{ab}^*\mathtt{c}$

Definition: Given a regular expression e, we can recursively define the language it recognizes $\mathcal{E}(e)$ using:

- $\mathcal{E}(0) \stackrel{def}{=} \emptyset$
- $\mathcal{E}(1) \stackrel{def}{=} \{\varepsilon\}$
- $\mathcal{E}(\mathtt{a}) \stackrel{def}{=} \{\mathtt{a}\}$
- $\mathcal{E}((e)) \stackrel{def}{=} \mathcal{E}(e)$
- $\mathcal{E}(e^*) \stackrel{def}{=} \mathcal{E}(e)^*$
- $\mathcal{E}(e^+) \stackrel{def}{=} \mathcal{E}(e)^+$
- $\mathcal{E}(e_1+e_2) \stackrel{def}{=} \mathcal{E}(e_1) \cup \mathcal{E}(e_2)$
- $\mathcal{E}(e_1e_2) \stackrel{def}{=} \mathcal{E}(e_1)\mathcal{E}(e_2)$

Let us consider the languages generated by the previous examples:

- $\mathcal{E}((\mathtt{a}+\mathtt{b})^*)$: Note that $\mathcal{E}(\mathtt{a}+\mathtt{b}) = \{\mathtt{a},\mathtt{b}\}$. Thus the language accepted by the expression is $\{\mathtt{a},\mathtt{b}\}^*$ — which is the set of all strings over the alphabet $\{\mathtt{a},\mathtt{b}\}$.

- $\mathcal{E}(\texttt{a}^*\texttt{b}^*)$: By definition, $\mathcal{E}(\texttt{a}^*)$ is the set of all (possibly empty) sequences of a. Hence, the expression accepts strings of the form $\texttt{a}^n$ or $\texttt{b}^n$.
- $\mathcal{E}((\texttt{ab}+\texttt{ba})^+)$ is the set of all strings built from sequences of ab or ba.
- $\mathcal{E}(\texttt{ab}^*\texttt{c}) = \{\texttt{ab}^n\texttt{c} \mid n \geq 0\}$

Although these language derivations have been presented informally, they can very easily be proved formally from the definitions just given.

5.2 Regular Grammars and Regular Expressions

One immediate result of these definitions and the theorems we have already proved about regular languages, is that regular grammars are at least as expressive as regular expressions. In other words, every language generated by a regular expression is regular.

Proposition: For any regular expression e, $\mathcal{E}(e)$ is a regular language.

Proof: The proof proceeds by structural induction over the syntax of expression e:

Base cases:

- $\mathcal{E}(0)$ is a regular language since there is a regular grammar producing it (namely a regular grammar with no production rules)
- $\mathcal{E}(1)$ is a regular language, since it is generated by $\langle \Sigma, \{S\}, \{S \to \varepsilon\}, S\rangle$, which is a regular grammar.
- For any terminal symbol a, $\mathcal{E}(\texttt{a})$ is a regular language, produced by $\langle \{\texttt{a}\}, \{S\}, \{S \to \texttt{a}\}, S\rangle$.

Inductive cases: Assume that the property holds of regular expressions e_1 and e_2 (that is, for both of them $\mathcal{E}(e_i)$ is a regular language).

- $\mathcal{E}((e_1))$ is the same as $\mathcal{E}(e_1)$ and is thus a regular language.

- $\mathcal{E}(e_1^*)$ is defined to be $(\mathcal{E}(e_1))^*$. But we have proved that for any regular language L, L^* is also a regular language. Thus $(\mathcal{E}(e_1))^*$ is a regular language and hence so is $\mathcal{E}(e_1^*)$.

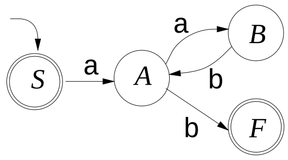
- As for $\mathcal{E}(e_1^+)$, simply note that given any regular language L, L^+ is also regular.

- Similarly, the union of two regular languages is also regular. Hence, $\mathcal{E}(e_1 + e_2)$ is a regular language.

- Finally, the catenation of two regular languages is also regular. Thus, by definition, $\mathcal{E}(e_1 e_2)$ is a regular language.

□

Example: Construct a deterministic finite state automaton which accepts the regular expression: $1 + \mathtt{a}(\mathtt{ab})^*\mathtt{b}$:



It is usually quite easy to construct a deterministic automaton, however, it is also easy to make a simple mistake. The theorems we already presented should ideally be used to construct the automata. Thus, for example, in the above example, we start by constructing regular grammars for the basic regular expressions (a, b, 1) using the method described in the last theorem. We would then proceed to construct the regular grammar for the union of a and b, which we then use to construct another grammar recognizing its Kleene closure $(\mathtt{a} + \mathtt{b})^*$. We then use the catenation theorem to add prefix a and postfix b and finally add ε to the language (or use the union theorem to add 1. This grammar is then converted to a DFSA via a NFSA.

As in other cases, this looks like an awful lot of boring work! However, it is important to realize that this can all be easily automated.

The next question arising is whether there are regular languages which cannot be expressed as regular expressions. This was shown not to be the case by Kleene. This, together with the previous proposition imply that regular languages and regular expressions are equally expressive.

Theorem: Every regular language can be expressed as a regular expression.

Strategy: The idea is to start off from a DFSA (recall that regular grammars and DFSA recognize exactly the same class of languages). This is decomposed into a number of automata, each of which has exactly one final state. We then prove (non-trivially) that such automata can be expressed as regular expressions.

Proof: Let L be a regular grammar. Then, there is a deterministic finite state automaton M, where $\mathcal{T}(M) = L$, where $M = \langle K, T, t, k_1, F \rangle$:

$$\begin{aligned} K &= \{k_1, k_2, \ldots k_n\} \\ F &= \{k_{\lambda_1}, k_{\lambda_2}, \ldots k_{\lambda_m}\} \end{aligned}$$

Now consider the machines:

$$M_i = \langle K, T, t, k_1, \{k_{\lambda_i}\} \rangle$$

Clearly, $\mathcal{T}(M) = \bigcup_{i=1}^{\lambda_m} \mathcal{T}(M_i)$.

Thus, if we can construct a regular expression e_i for each M_i, we have:

$$\begin{aligned} & \mathcal{E}(e_1 + \ldots e_{\lambda_m}) \\ = \; & \mathcal{E}(e_1) + \ldots \mathcal{E}(e_{\lambda_m}) \\ = \; & \bigcup_{i=1}^{\lambda_m} \mathcal{T}(M_i) \\ = \; & \mathcal{T}(M) \\ = \; & L \end{aligned}$$

and thus L can also be expressed as a regular expression.

Let us consider one particular M_i and rename the states such that:

$$\begin{aligned} K &= \{k_1, k_2, \ldots k_n\} \\ F &= \{k_n\} \end{aligned}$$

We now define language $T^l_{i,j}$ (where $1 \leq i, j \leq n$ and $0 \leq l \leq n$) to be the set of all strings x satisfying:

1. $t_*(k_i, x) = k_j$ and
2. for all proper prefixes y of x, $t_*(k_i, y) = \{k_m \mid m \leq l\}$

Informally, $T^l_{i,j}$ is thus the set of all strings which send automaton M_i from k_i to k_j by only going through states in $\{k_1 \dots k_l\}$.

The proof proceeds by induction on l, to show that $\mathcal{T}(T^l_{i,j}$ is a regular language.

Base case ($l = 0$): We split this part into two cases: $i = j$ and $i \neq j$.

- $i = j$: $T^0_{i,j} = \mathcal{E}(1)$ if there is no a such that $t(k_i, a) = k_j$. Otherwise $T^0_{i,j}$ is simply the union of all such labels: $1 + a_1 + a_2 + \ldots + a_m$. In both cases, $T^0_{i,j}$ is equivalent to a regular expression.
- $i \neq j$: $T^0_{i,j} = \mathcal{E}(0)$ if there is no a such that $t(k_i, a) = k_j$. Otherwise $T^0_{i,j}$ is simply the union of all such labels: $a_1 + a_2 + \ldots + a_m$. In both cases, $T^0_{i,j}$ is equivalent to a regular expression.

Inductive case: Assume $T^l_{i,j}$ can always be expressed as a regular expression.

We now claim that $T^{l+1}_{i,j} = T^l_{i,j} + T^l_{i,l+1}(T^l_{l+1,l+1})^* T^l_{l+1,j}$.

Informally, it says that the strings which take the automaton from state t_i to state t_j through states in $\{k_1, \dots k_{l+1}\}$ either:

- do not go through k_{l+1} and hence are also in $T^l_{i,j}$, or
- go to k_{l+1} a number of times. In between these 'visits' k_{l+1} is not traversed again. Thus, the overall trip can be expressed as a sequence of trips from k_i to k_{l+1}, from k_{l+1} to k_{l+1} for a number of times, and finally from k_{l+1} to k_j, where in each of these mini-trips, k_{l+1} is not traversed. Hence, the whole trip is in $T^l_{i,l+1}(T^l_{l+1,l+1})^* T^l_{l+1,j}$.

By the inductive hypothesis, all the traversals can now be expressed as regular expressions and thus, so can $T^l_{i,j}$.

This completes the induction proof.

Now, simply note that $\mathcal{T}(M_i) = T^n_{1,n}$. Hence, $\mathcal{T}(M_i)$ can be expressed as a regular expression, and so can $\mathcal{T}(M)$.

$\square$

5.3 Exercises

1. Using the results of the theorems proved, construct a DFSA which recognizes strings described by the regular expression $\mathtt{a}(\mathtt{a} + \mathtt{9})^*$.

2. Prove some of the following laws about regular expressions:

   Identity laws: 0 is the identity over + and 1 over catenation.

   (a) $e + 0 = 0 + e = e$
   (b) $e1 = 1e = e$

   Zero law: 0 is the zero over catenation.

   (a) $e0 = 0e = 0$

   Associativity laws: Both choice and catenation are associative.

   (a) $(ef)g = e(fg) = efg$
   (b) $(e + f) + g = e + (f + g) = e + f + g$

   Commutativity laws: Both choice and catenation are commutative.

   (a) $e + f = f + e$

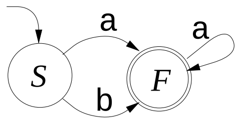
   Distributivity laws: Choice distributes over catenation.

   (a) $e(f + g) = ef + eg$
   (b) $(e + f)g = eg + fg$

   Closure Laws: The Kleene closure operator obeys the following laws:

   (a) $e^* = 1 + e^+$
   (b) $0^* = 0^+ = 0$
   (c) $1^* = 1^+ = 1$

3. Using Kleene's theorem, construct a regular expression which describes the language accepted by the following DFSA:



Use the laws in the previous question to prove that this expression is equivalent to $(\mathtt{a} + \mathtt{b})\mathtt{a}^*$.

5.4 Conclusions

The constructive result we had proved, that regular grammars are just as expressive as DFSA, and the ease of implementation of DFSA indicated the interesting possibility of implementing a program which, given a regular grammar, constructs a DFSA to be able to efficiently deduce whether a given string is in the grammar or not. The question left pending was how to describe the regular grammar. The only possibility we had was to somehow encode the production rules as text.

The first result in this chapter opened a new possibility. All regular expressions are in fact regular languages and we have the means to construct a regular grammar from such an expression. Can we use regular expressions to describe regular languages? It is rather obvious that such a representation is much more 'ascii-friendly', but are we missing out on something? Are there regular languages which are not expressible as regular expressions? The second theorem rules this out, justifying the use of regular expressions as the input format of regular languages.

This is precisely what LEX does. Given a regular expression, LEX automatically generates the code to recognize whether a string is in the expression or not and all this is done via DFSA as discussed in this course. You will be studying further what LEX does in next year's course on 'Compiling Techniques'.

The course has, up to this point shown the equivalence of:

- Languages accepted by regular grammars

- Languages recognizable by non-deterministic finite state automata
- Languages recognizable by deterministic finite state automata
- Languages describable by regular expressions

However, we have mentioned that certain context free languages cannot be recognized by a regular grammar. In particular, there is no way of writing a regular grammar which recognizes matching parentheses. This severely limits the use of regular grammars in compiler writing. We thus now try to replicate similar work to that we have just done on regular languages but on context free languages. Is there a machine which we can construct from a context free grammar which easily checks whether a string is accepted or not? Can this conversion be automated? These are the questions we will now be asking.

If you are wondering what is the use of LEX in compiler writing if LEX recognizes only regular languages and regular languages are not expressive enough in compiler writing, the answer is that compilers perform at least two passes over the text they are given. The first, so called *lexical analysis* simply identifies tokens such as `while` or `:=`. The job of the lexical analyzer is to group together such symbols to simplify the work of the *parser*, which checks the structure of the tokens (for example `while if` would make no sense in most languages). The lexical analysis can usually be readily performed by a regular grammar, whereas the parser would need to use a context-free language.

Chapter 6

Pushdown Automata

The main problem with finite state automata, is that they have no way of 'remembering'. Pushdown automata are basically an extension of finite state automata, which allows for memory. The extra information is carried around in the form of a stack.

6.1 Stacks

A stack is a standard data structure which is used to store information to be retrieved at a later stage. A stack has an underlying data type Σ, such that all information placed on the stack if of that type. Two operations can be used on a stack:

Push: The function *push* takes a stack of underlying type Σ and a value of type Σ and returns a new stack which is identical to the original but with the given value on top.

Pop: The function *pop* takes a non-empty stack and returns the value on top of the stack. The stack is also returned with the top value removed.

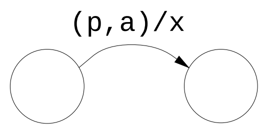
We will use strings to formally define stacks. ε is the empty stack, whereas any string over Σ is a possible configuration of a stack with underlying type Σ. The functions on stacks are formalized as follows:

$$
\begin{array}{rcl}
push & : & Stack_\Sigma \times \Sigma \rightarrow Stack_\Sigma \\
push(s, a) & \stackrel{def}{=} & sa
\end{array}
$$

$$
\begin{array}{rcl}
pop & : & Stack_\Sigma \rightarrow \Sigma \times Stack_\Sigma \\
pop(as) & \stackrel{def}{=} & (a, s)
\end{array}
$$

6.2 An Informal Introduction

A pushdown automaton has a stack which it uses to store information. Upon initialization, the stack always starts off with a particular value being pushed (normally used to act as a marker that the stack is about to empty). Every transition now depends not only on the input, but also on the value on the top of the stack. Upon performing a transition, a number of values may also be pushed on the stack. In diagrams, transitions are now marked as shown in the figure below:

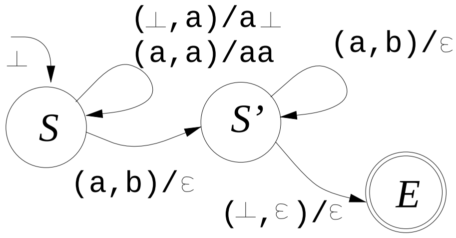


The label $(\mathtt{p}, \mathtt{a})/x$ is interpreted as follows: pop a value off the stack, if the symbol just popped is `p` and the input is `a`, then perform the transition and push onto the stack the string x. If the transition cannot take place, try another using the value just popped. The machine 'crashes' if it tries to pop a value off an empty stack.

The arrow marking the initial state is marked by a symbol which is pushed onto the stack at initialization.

As in the case of FSA, we also have a number of final states which are used to determine which strings are accepted. A string is accepted if, starting the machine from the initial state, it terminates in one of the final states.

Example: Consider the PDA below:



Initially, the value $\perp$ is pushed onto the stack. This will be called the 'bottom of stack marker'. While we are at the initial state S, read an input. If it is `a` push back to the stack whatever has just been popped, together with an extra `a`. In other words, the stack now contains as many `a`s as have been accepted.

When a `b` is read, the state now changes to S'. The top value of the stack (`a`) is also discarded. While in S' the PDA continues accepting `b`s as long as there are `a`s on the stack. When finally, the bottom of stack marker is encountered, the machine goes to the final state E.

Hence, the machine is accepting all strings in the language $\{\mathtt{a}^n\mathtt{b}^n \mid n \geq 1\}$. Note that this is also the language of the context free grammar with production rules: $S \rightarrow \mathtt{ab} \mid \mathtt{a}S\mathtt{b}$.

Recall that it was earlier stated that this language cannot be accepted by a regular grammar. This seems to indicate that we are on the right track and that pushdown automata accept certain languages for which no finite state automaton can be constructed to accept.

6.3 Non-deterministic Pushdown Automata

Definition: A non-deterministic pushdown automaton M is a 7-tuple:

$$M = \langle K, T, V, P, k_1, A_1, F \rangle$$

$$
\begin{array}{lcl}
K & = & \text{a finite set of states} \\
T & = & \text{the input alphabet of } M \\
V & = & \text{the stack alphabet} \\
P & = & \text{transition function of } M \text{ with type:} \\
 & & (K \times V \times (T \cup \{\varepsilon\}) \rightarrow \mathbb{P}(K \times V^*) \\
k_1 & \in & K \text{ is the start state} \\
A_1 & \in & V \text{ is the initial symbol placed on the stack} \\
F & \subseteq & K \text{ is the set of final states}
\end{array}
$$

Of these, the production rules (P) may need a little more explaining. P is a total function such that $P(k, v, i) = \{(k'_1, s_1), \ldots (k'_n, s_n)\}$, means that: in state k, with input i (possibly ε meaning that the input tape is not inspected) and with symbol v on top of the stack (which is popped away), M can evolve to any of a number of states k'_1 to k'_n. Upon transition to k'_i, the string s_i is pushed onto the stack.

Example: The PDA already depicted is formally expressed by:

$$
M \quad = \quad \langle \{S, S', E\}, \{\mathtt{a}, \mathtt{b}\}, \{\mathtt{a}, \mathtt{b}, \perp\}, P, S, \perp, \{E\} \rangle
$$

where P is defined as follows:

$$
\begin{array}{rcl}
P(S, \perp, \mathtt{a}) & = & \{(S, \mathtt{a}\perp)\} \\
P(S, \mathtt{a}, \mathtt{a}) & = & \{(S, \mathtt{aa})\} \\
P(S, \mathtt{a}, \mathtt{b}) & = & \{(S', \varepsilon)\} \\
P(S', \mathtt{a}, \mathtt{b}) & = & \{(S', \varepsilon)\} \\
P(S, \perp, \varepsilon) & = & \{(E, \varepsilon)\} \\
P(X, x, y) & = & \emptyset \text{ otherwise}
\end{array}
$$

Note that this NPDA has no non-deterministic transitions, since every state, input, stack value triple has at most one possible transition.

Whereas before, the current state described all the information needed about the machine to be able to deduce what it can do, we now also need the current state of the stack. This information is called the *configuration* of the machine.

Definition: The set of configurations of M is the set of all pairs (state, string): $K \times V^*$.

We would now like to define which configurations are reachable from which configurations. If there is a final state in the set of configurations reachable from (k_1, A_1) with input s, s would then be a string accepted by M.

Definition: The transition function of a PDA M, written t_M is a function, which given a configuration and an input symbol, returns the set of states in which the machine may terminate with that input.

$$
\begin{aligned}
t_M &: (K \times V^*) \times (T \cup \{\varepsilon\}) \to K \times V^* \\
t_M((k, xX), a) &\stackrel{def}{=} \{(k', yX) \mid (k', y) \in P(k, x, a)\}
\end{aligned}
$$

Definition: The string closure transition function of a PDA M, written t_M^* is the function, which given an initial configuration and input string, returns the set of configurations in which the automaton may end up in after using up the input string.

$$
t_M^* : (K \times V^*) \times (T \cup \{\varepsilon\}) \to \mathbb{P}(K \times V^*)
$$

We start by defining the function for an empty stack:

$$
\begin{aligned}
t_M^*((k, \varepsilon), \varepsilon) &\stackrel{def}{=} \{(k, \varepsilon)\} \\
t_M^*((k, \varepsilon), a) &\stackrel{def}{=} \emptyset
\end{aligned}
$$

For a non-empty stack, and input $a_1 \ldots a_n$, (where each $a_i \in T \cup \{\varepsilon\}$, we need to find

$$
\begin{aligned}
t_M^*((k, X), w) \stackrel{def}{=} \; & \{(k', X') \mid \\
& \exists a_1, \ldots a_n : T \cup \{\varepsilon\} \cdot w = a_1 \ldots a_n \\
& \quad \exists c_1, \ldots c_n : K \times V^* \cdot \\
& \quad\quad c_1 \in t_M((k, X), a_1) \\
& \quad\quad c_{i+1} \in t_M(c_i, a_i) \text{ for all } i \\
& \quad\quad (k', X') \in t_M(c_n, a_n)
\end{aligned}
$$

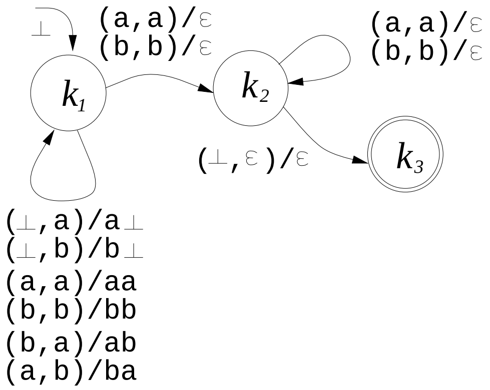
We say that $c' = (k', X')$ is reachable from $c = (k, X)$ by a if $c' \in t^*_M(c, a)$.

Definition: The *language* recognized by a non-deterministic pushdown automaton $M = \langle K, T, V, P, k_1, A_1, F\rangle$, written as $\mathcal{T}(M)$ is defined by:

$$\mathcal{T}(M) \quad \stackrel{def}{=} \quad \{x \mid x \in T^*,\ (F \times V^*) \cap t^*_M((k_1, A_1), x) \neq \emptyset\}$$

Informally, there is at least one configuration with a final state reachable from the initial configuration with x.

Example: Consider the following NPDA:



This automaton accepts non-empty even-length palindromes over `a`, `b`.

Now consider the acceptance of `baab`:

$$\begin{array}{ccc}
 & (k_1, \bot) & \\
 & \downarrow \texttt{b} & \\
 & (k_1, \texttt{b}\bot) & \\
 & \downarrow \texttt{a} & \\
 & (k_1, \texttt{ab}\bot) & \\
\swarrow \texttt{a} & & \searrow \texttt{a} \\
(k_1, \texttt{aab}\bot) & & (k_2, \texttt{b}\bot) \\
\downarrow \texttt{b} & & \downarrow \texttt{b} \\
(k_1, \texttt{baab}\bot) & & (k_2, \bot) \\
 & & \downarrow \varepsilon \\
 & & (k_3, \varepsilon)
\end{array}$$

From $(k_1, \perp)$, we can only reach $(k_1, \mathtt{b}\perp)$ with $\mathtt{b}$, from where we can only reach $(k_1, \mathtt{ab}\perp)$ with $\mathtt{a}$. Now, with input $\mathtt{a}$, we have a choice. We can either reach $(k_1, \mathtt{aab}\perp)$ or $(k_2, \mathtt{b}\perp)$. In terms of the machine, it may be seen as if it is not yet clear whether we have started reversing the word or whether we are still giving the first part of the string. The tree above shows how the different alternatives branch up the tree.

Clearly, $t^*_M((k_1, \perp), \mathtt{baab}) = \{(k_1, \mathtt{baab}\perp), (k3, \varepsilon)\}$.

Since $t^*_M((k_1, \perp), \mathtt{baab}) \cap (F \times V^*) \neq \emptyset$, we conclude that $\mathtt{baab} \in \mathcal{T}(M)$.

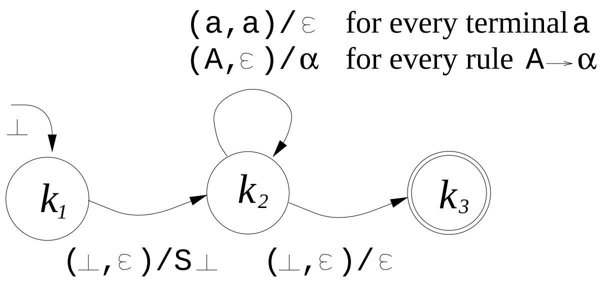
6.4 Pushdown Automata and Languages

The whole point of defining these pushdown automata was to see whether we can define a class of machines which are capable of recognizing the class of context free languages. In this section, we prove that non-deterministic pushdown automata recognize exactly these languages. Again, the proofs are constructive in nature and thus allow us to generate automata from grammars and vice-versa.

6.4.1 From CFLs to NPDA

Theorem: For every context free language L there is a non-deterministic pushdown automaton M such that $\mathcal{T}(M) = L$.

Strategy: The trick is to have at the top of the stack the currently active non-terminal. For every rule $A \rightarrow \alpha$ we would then add a transition rule which is activated if A is on the top of the stack, and which replaces A by α. We also add transition rules for every terminal symbol which work by matching input with stack contents. The whole automaton will be made up of just 3 states, as shown in the diagram below:



Proof: Let $G = \langle \Sigma, N, P, S \rangle$ be a context free grammar such that $\mathcal{L}(G) = L$. We now construct a NPDA:

$$
\begin{aligned}
M \quad = \langle \quad & \{k_1, k_2, k_3\}, \\
& \Sigma, \\
& \Sigma \cup N \cup \{\bot\}, \\
& P_M, \\
& k_1, \\
& \bot, \\
& \{k_3\} \rangle
\end{aligned}
$$

where P_M is:

- From the initial state k_1 we push S and go to k_2 immediately.

$$P((k_1, \bot), \varepsilon) \quad = \quad (k_2, S\bot)$$

- Once we reach the bottom of stack marker in k_2 we can terminate.

$$P((k_2, \bot), \varepsilon) \quad = \quad (k_3, \varepsilon)$$

- For every non-terminal A in N with at least one production rule in P of the form $A \to \alpha$ we add the rule:

$$P((k_2, A), \varepsilon) \quad = \quad \{(k_2, \alpha) \mid A \to \alpha \in P\}$$

- For every terminal symbol a in Σ we also add:

$$P((k_2, a), a) \quad = \quad \{(k_2, \varepsilon)\}$$

We now prove that $\mathcal{T}(M) = L$.

The proof is based on two observations:

1. $S \overset{*}{\Rightarrow}_G x\alpha$ is a leftmost derivation $\Leftrightarrow$ $(k_2, \alpha\perp) \in t((k_1, \perp), x)$

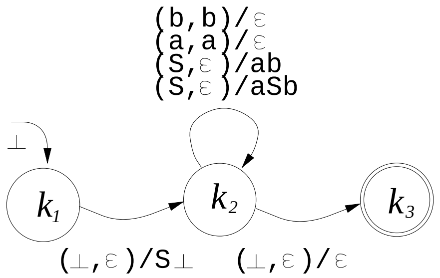
   This can be proved by induction on the length of the derivation or production.

2. $(t_3, \alpha) \in t((k_1, \perp), x) \Rightarrow \alpha = \varepsilon$

$$\begin{array}{ll} & x \in L \\ \Leftrightarrow & x \in \mathcal{L}(G) \\ \Leftrightarrow & S \overset{*}{\Rightarrow}_G x \\ \Leftrightarrow & (k_2, \perp) \in t((k_1, \perp), x) \\ \Leftrightarrow & (k_3, \varepsilon) \in t((k_1, \perp), x) \\ \Leftrightarrow & (k_3 \times V^*) \cap t((k_1, \perp), x) \neq \emptyset \\ \Leftrightarrow & (F \times V^*) \cap t((k_1, \perp), x) \neq \emptyset \\ \Leftrightarrow & x \in \mathcal{T}(M) \end{array}$$

$\square$

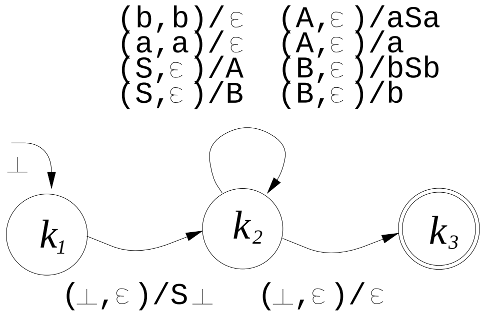
Example: Consider the grammar $G = \langle \{\mathtt{a}, \mathtt{b}\}, \{S\}, \{S \rightarrow \mathtt{ab} \mid \mathtt{a}S\mathtt{b}\}, S \rangle$. From this we can construct the PDA:



Example: Consider a slightly more complex example, with grammar $G = \langle\{\mathtt{a},\mathtt{b}\}, \{S, A, B\}, P, S\rangle$, where P is:

$$\begin{array}{rcl} S & \rightarrow & A \mid B \\ A & \rightarrow & \mathtt{a}S\mathtt{a} \mid \mathtt{a} \\ B & \rightarrow & \mathtt{b}S\mathtt{b} \mid \mathtt{b} \end{array}$$

Clearly, G recognizes all palindromes over `a` and `b` except for ε. Constructing a NPDA with the same language we get:



6.4.2 From NPDA to CFGs

We would now like to prove the inverse: that languages produced by NPDA are all context free languages. Before we prove the result, we prove a lemma which we will use in the proof.

Lemma: For every NPDA M, there is an equivalent NPDA M' such that:

1. M' has only one final state
2. the final state is the only one in which the state can be clear
3. the automaton clears the stack upon termination

Strategy: The idea is to add a new bottom of stack marker which is pushed onto the stack before M starts. Whenever M terminates we go to a new state which can clear the stack.

Proof: Let $M = \langle K, T, V, P, k_1, A_1, F\rangle$.

Now construct $M' = \langle K', T, V', P', k_1', \perp, F' \rangle$:

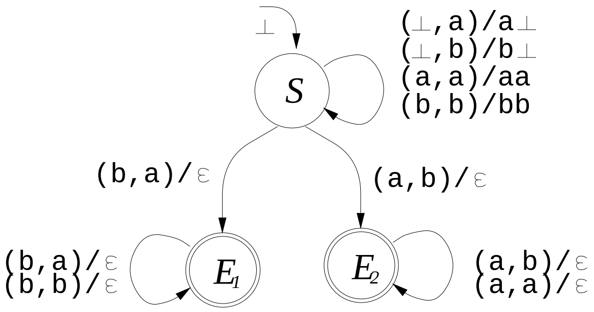
$$\begin{array}{rcl} K' & = & K \cup \{k_1', k_2'\} \\ V' & = & K \cup \{\perp\} \\ P' & = & \quad P \\ & & \cup \quad \{((k_1', \perp), \varepsilon) \rightarrow \{(k_1, A_1 \perp)\}\} \\ & & \cup \quad \{((k, A), \varepsilon) \rightarrow \{(k_2', \varepsilon)\} \mid k \in F, A \in V'\} \\ & & \cup \quad \{((k_2', A), \varepsilon) \rightarrow \{(k_2', \varepsilon)\} \mid A \in V'\} \\ F' & = & \{k_2'\} \end{array}$$

Note that throughout the execution of M, the stack can never be empty, since there will be, at least $\perp$ left on the stack.

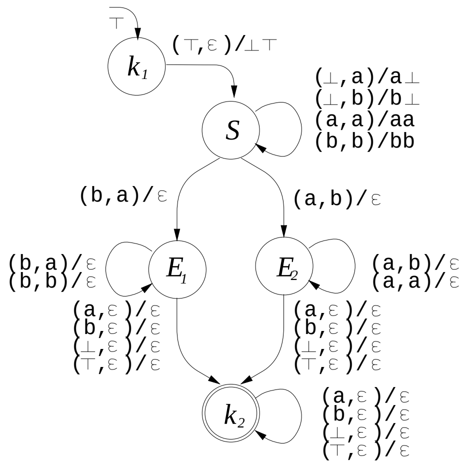
The proof will not be taken any further here. Check in the textbooks if you are interested in how the proof is then completed.

$\square$

Example: Given the NPDA below we want to construct a NPDA satisfying the constraints motioned in the lemma.



Following the method given in the lemma, we add two new states and a new symbol in the stack alphabet. The resultant machine looks like:



Theorem: For any NPDA M, $\mathcal{T}(M)$ is a context free language.

Strategy: The idea is to construct a grammar, where, for each $A \in V$ (the stack alphabet), we have a family of non-terminals A^{ij}. Each A^{ij} generates the input strings which take M from state k_i to k_j, and remove A from the top of the stack. A_1^{1n} is the non-terminal which takes M from k_1 to k_n removing A_1 off the stack, which is exactly what is desired.

Proof: Assume that M satisfies the conditions of the previous lemma (otherwise obtain a NPDA equivalent to M with these properties, as described earlier). Label the states k_1 to k_n, where k_1 is the initial state, and k_n is the (only) final one. Then:

$$M = \langle \{k_1, \ldots k_n\}, T, V, P, k_1, A_1, \{k_n\} \rangle$$

We now construct a context-free grammar G such that $\mathcal{L}(G) = \mathcal{T}(M)$.

Let $G = \langle \Sigma, N, R, S \rangle$.

$$\begin{array}{rcl}
N & = & \{A^{ij} \mid A \in V,\ 1 \leq i, j \leq n\} \\
S & = & A_1^{1n} \\
R & = & \{A^{in_m} \to aB_1^{jn_1} B_2^{n_1 n_2} \ldots B_m^{n_{m-1} n_m} \mid \\
& & \quad (k_j, B_1 B_2 \ldots B_m) \in t((k_i, A), a),\ 1 \leq n_1, n_2, \ldots n_m \leq n\} \cup \\
& & \{A^{ij} \to a \mid (k_j, \varepsilon) \in t((k_i, A), a)\}
\end{array}$$

The construction of R is best described as follows: Whenever M can evolve from state k_i (with A on the stack) and input a to k_j (with $B_1 \ldots B_m$ on the stack), we add rules of the form:

$$A^{in_m} \to aB_1^{jn_1} B_2^{n_1 n_2} \ldots B_m^{n_{m-1} n_m}$$

(with arbitrary n_1 to n_m). This says that M may evolve from k_i to k_{n_m} by reading a and thus evolving to k_j with $B_1 \ldots B_m$ on the stack. After this, it is allowed to travel around the states — as long as it removes all the Bs and ends in state k_m.

For every transition from k_i to k_j (removing A off the stack) and with input a, we also add the production rule $A^{ij} \to a$.

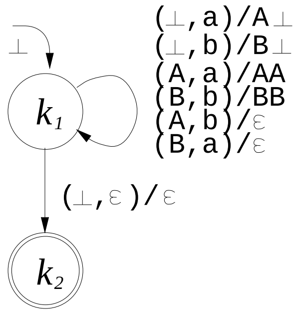
From this construction:

$$\begin{array}{ll} & x \in \mathcal{T}(M) \\ \Leftrightarrow & (k_n, \varepsilon) \in t_M^*((k_1, A_1), x) \\ \Leftrightarrow & A_1^{1n} \stackrel{*}{\Rightarrow}_G x \\ \Leftrightarrow & x \in \mathcal{L}(G) \end{array}$$

Hence $\mathcal{T}(M) = \mathcal{L}(G)$.

□

Example: Consider the following NPDA:



This satisfies the conditions placed, and therefore, we do not need to apply the construction from the lemma. The set of strings this automaton generates are ones ranging over `a` and `b`, where the count of `a` is the same as the count of `b`.

To obtain a context free grammar for this NPDA, we start by listing the non-empty instances of the transition function:

$$\begin{array}{rcl}
t((k_1,\perp),\mathtt{a}) & = & \{(k_1, A\perp)\}\\
t((k_1,\perp),\mathtt{b}) & = & \{(k_1, B\perp)\}\\
t((k_1,A),\mathtt{a}) & = & \{(k_1, AA)\}\\
t((k_1,A),\mathtt{b}) & = & \{(k_1, \varepsilon)\}\\
t((k_1,B),\mathtt{a}) & = & \{(k_1, \varepsilon)\}\\
t((k_1,B),\mathtt{b}) & = & \{(k_1, BB)\}\\
t((k_1,\perp),\varepsilon) & = & \{(k_2, \varepsilon)\}
\end{array}$$

From each of these we generate a number of production rules. This is demonstrated on the first, fourth and last line from the equations above. Note that we have three stack symbols (A, B and $\perp$) and two states (k_1 and k_2). We will therefore have twelve non-terminal symbols:

$$\{\perp^{11},\perp^{12},\perp^{21},\perp^{22},A^{11},A^{12},A^{21},A^{22},B^{11},B^{12},B^{21},B^{22}\}$$

where $\perp^{12}$ is the start symbol.

First line: $t((k_1,\perp),\mathtt{a}) = \{(k_1, A\perp)\}$

In this case, we have an instance of starting with $\perp$ on the stack and going from state 1 to state 1. The non-terminal appearing on the left hand side of the rules will thus be $\perp^{1i}$. The input read is $\mathtt{a}$ and thus the rules will read $\perp^{1i} \to \mathtt{a}\alpha$. The application also leaves $A\perp$ on the stack, which have to be removed: $\perp^{1i} \to \mathtt{a}A^{1j}\perp^{ji}$. The rules introduced are thus:

$$\perp^{11} \to \mathtt{a}A^{11}\perp^{11} \mid \mathtt{a}A^{12}\perp^{21}$$

$$\perp^{12} \to \mathtt{a}A^{11}\perp^{12} \mid \mathtt{a}A^{12}\perp^{22}$$

Fourth line: $t((k_1,A),\mathtt{b}) = \{(k_1, \varepsilon)\}$

Since this transition leaves the stack empty, we use the second rule to generate productions. We start off with A on the stack and go from state 1 to state 1. Hence we are defining A^{11}. Since upon reading $\mathtt{b}$, we are leaving the stack empty, we get the rule:

$$A^{11} \to \mathtt{b}$$

Last line: $t((k_1, \perp), \varepsilon) = \{(k_2, \varepsilon)\}$

The reasoning is identical to the previous case, except that we are now not reading any input.

$$A^{12} \to \varepsilon$$

The complete set of production rules is:

$$\begin{array}{lcl}
\perp^{11} & \to & \mathtt{a}A^{11}\perp^{11} \mid \mathtt{a}A^{12}\perp^{21} \mid \mathtt{b}B^{11}\perp^{11} \mid \mathtt{b}B^{12}\perp^{21} \\
\perp^{12} & \to & \mathtt{a}A^{11}\perp^{12} \mid \mathtt{a}A^{12}\perp^{22} \mid \mathtt{b}B^{11}\perp^{12} \mid \mathtt{b}B^{12}\perp^{22} \mid \varepsilon \\
A^{12} & \to & \mathtt{a}A^{11}A^{12} \mid \mathtt{a}A^{12}A^{22} \\
A^{11} & \to & \mathtt{a}A^{11}A^{11} \mid \mathtt{a}A^{12}A^{21} \mid \mathtt{b} \\
B^{12} & \to & \mathtt{b}B^{11}B^{12} \mid \mathtt{b}B^{12}B^{22} \\
B^{11} & \to & \mathtt{b}B^{11}B^{11} \mid \mathtt{b}B^{12}B^{21} \mid \mathtt{a}
\end{array}$$

We can remove states like $\perp^{22}$ since they cannot evolve any further (no outgoing arrows) to get:

$$\begin{array}{lcl}
\perp^{11} & \to & \mathtt{a}A^{11}\perp^{11} \mid \mathtt{b}B^{11}\perp^{11} \\
\perp^{12} & \to & \mathtt{a}A^{11}\perp^{12} \mid \mathtt{b}B^{11}\perp^{12} \mid \varepsilon \\
A^{12} & \to & \mathtt{a}A^{11}A^{12} \\
A^{11} & \to & \mathtt{a}A^{11}A^{11} \mid \mathtt{b} \\
B^{12} & \to & \mathtt{b}B^{11}B^{12} \\
B^{11} & \to & \mathtt{b}B^{11}B^{11} \mid \mathtt{a}
\end{array}$$

Now we note that from the initial state $\perp^{12}$ only B^{11} and A^{11} are reachable. Copying and renaming the non-terminal symbols, we get:

$$\begin{array}{lcl}
S & \to & \mathtt{a}AS \mid \mathtt{b}BS \mid \varepsilon \\
A & \to & \mathtt{a}AA \mid \mathtt{b} \\
A & \to & \mathtt{b}BB \mid \mathtt{a}
\end{array}$$

6.5 Exercises

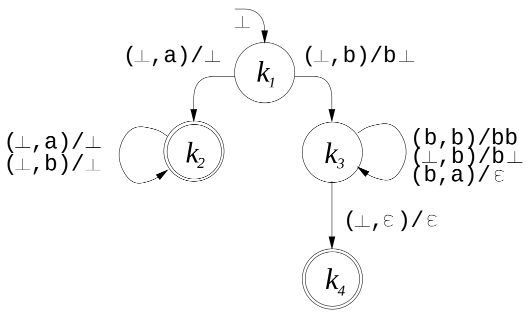
1. Construct an NPDA to recognize valid strings in the language described using the BNF notation (with initial non-terminal $\langle program \rangle$):

$$\begin{array}{lcl}
\langle var \rangle & ::= & \texttt{a} \mid \texttt{b} \\
\langle val \rangle & ::= & \texttt{0} \mid \texttt{1} \\
\langle skip \rangle & ::= & \mathbb{I} \\
\langle assign \rangle & ::= & \langle var \rangle\texttt{=}\langle var \rangle \mid \langle var \rangle\texttt{=}\langle val \rangle \\
\langle instr \rangle & ::= & \langle assign \rangle \mid \langle skip \rangle \\
\langle program \rangle & ::= & \langle instr \rangle \mid \langle instr \rangle\texttt{;}\langle program \rangle
\end{array}$$

   Expand the grammar to deal with:

   - Program blocks with `{` and `}` used to begin and end blocks (respectively)
   - Conditionals, of the form $\langle program\text{-}block \rangle \triangleleft \langle var \rangle \triangleright \langle program\text{-}block \rangle$ ($P \triangleleft b \triangleright Q$ is read 'P if b, else Q).
   - While loops of the form $\langle var \rangle * \langle program\text{-}block \rangle$.

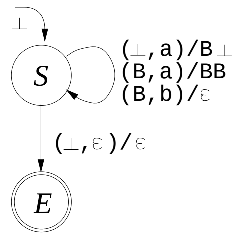
2. Consider the NPDA depicted below:



   (a) Describe the language recognized by this NPDA.

   (b) Calculate an equivalent NPDA with one final state.

3. Consider the following NPDA.



(a) Describe what strings it accepts.

(b) Construct a context free grammar which recognizes the same language as accepted by the automaton.

Chapter 7

Minimization and Normal Forms

7.1 Motivation

We have seen various examples in which distinct grammars or automata produce the same language. This property makes it difficult to compare grammars and automata without reverting back to the language they produce. Similarly, there is the issue of optimization, where, for automization reasons, we would like grammars to have as few non-terminals, and automata as few states as possible.

Imagine we are to discuss properties of triangles — any triangle in any position in a three dimensional space is to be studied. Clearly, we can define the set of all triangles to be the set of all 3-tuples of points referred to in cartesian notation (x, y, z). Obviously, there are other possible ways of defining the set of all triangles, but this is the one we choose.

Now, imagine that for a particular application, we are only interested in the length of the sides of a triangle. With this in mind, we define an equivalence relation on triangles, and say that a triangle t_1 is equivalent to a triangle t_2 if and only if the lengths of the sides of t_1 are the same as those of t_2. Note that this will include mirror image triangles to be considered as equivalent.

The process of checking the equivalence between two triangles needs considerable computation. However, we notice that moving triangles around space (including rotation, translation and flipping) does not change the triangle in any way (as far as our interests are concerned). We could move around any triangle we are given such that:

1. The longest side starts at the origin and extends in the positive x-axis direction.

2. If we call the point at the origin A, and the other point lying on the x-axis B, we flip the triangle such that AB is the second longest side, and B lies in the xy-plane.

If we call a triangle which satisfies these criteria, a normal form triangle, we notice that every triangle has an equivalent normal form triangle. Furthermore, for every triangle, there is only one such equivalent normal form triangle. This allows us to compare triangles just by comparing their normal forms (now two triangles are equivalent exactly when they have a common normal form) and certain operations may be simpler to perform on the normal form (for example, the area covered by a normal form triangle is simply half the x coordinate of the second point multiplied by the y coordinate of the third point).

Furthermore, in certain cases, checking the equivalence of two objects may be much more difficult than translating them into their normal forms and comparing the results. This was not the case in the restricted triangles example, since rotation of a triangle is much more computationally intensive than calculating the lengths of the sides.

This is the approach we would like to take to formal languages. If we can find a normal form (such that every grammar/automaton has one and only one equivalent grammar in normal form), we can then compare the normal forms of grammars rather than general grammars (which would be easier). A normal form is sometimes called a canonical form.

We are thus addressing two different questions in this chapter:

- Can we find a normal form for grammars and automata?

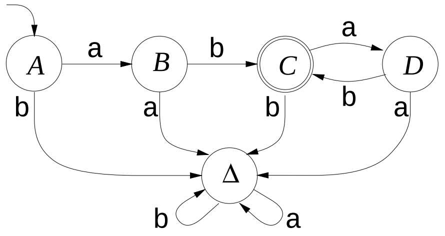
- Can we find an easy algorithm to minimize grammars and automata?

7.2 Regular Languages

We start off by considering the simpler class of regular languages. Clearly, by the theorem stating that finite state automata and regular grammars are equally expressive, the question will be the same whether we consider either of the two. In this case we will be considering deterministic finite state automata.

7.2.1 Overview of the Solution

Look at the total DFSA depicted below:



Essentially, the states are a means of defining an equivalence relationship on the input strings. Two strings are considered equivalent (with respect to this automaton) if and only, when the automaton is started up with either string, in both cases, it ends up in the same state. Thus, for example, the strings `aba` and `ababa` both send the automaton to state D and are thus related. Similarly, `b` and `aa` send the automaton from the start state to state Δ and are similarly related.

Equivalence relations define a partition of the underlying set. Thus, equivalence with respect to this automaton partitions the set of all strings over `a` and `b` into a number of parts (equal to the number of states, 5).

Clearly, we cannot define a single state automaton with the same language as given by this automaton. Does this mean that there is a lower limit to the number of meaningful partitions we can divide the set of all strings over an alphabet into?

It turns out that for every regular language there is such a constant. We can also construct a total DFSA with this number of states, and furthermore, it is unique (that is, there is no other DFSA with the same number of states but different connections between them which gives the same language). This completes our search for a minimal automaton and a canonical form for DFSA (and hence regular languages).

The strategy of the proof is thus:

1. Define equivalence with respect to a particular DFSA.
2. Show that the number of equivalence classes (partitions) for the particular language in question has a fixed lower bound.
3. Design a machine which recognizes the language in question with the minimal number of states.
4. Show that it is unique.

7.2.2 Formal Analysis

We will start by defining an equivalence relation on strings for any given language.

Definition: For a given language L over alphabet Σ, we define the relation $\equiv_L$ on strings over Σ:

$$\begin{array}{rcl} \equiv_L & \subseteq & \Sigma^* \times \Sigma^* \\ x \equiv_L y & \stackrel{def}{=} & \forall z : \Sigma^* \cdot xz \in L \Leftrightarrow yz \in L \end{array}$$

Proposition: $\equiv_L$ is an equivalence relation.

Example: Consider the language $L = (\mathtt{ab})^+$. By definition of the $\equiv_L$, we can show that $\mathtt{ab} \equiv_L \mathtt{abab}$. The proof would look like:

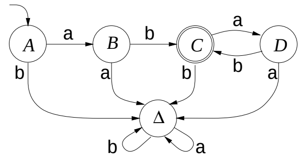
$$\begin{array}{rl} & \mathtt{ab}x \in L \\ \Leftrightarrow & \mathtt{ab}x \in (\mathtt{ab})^+ \\ \Leftrightarrow & x \in (\mathtt{ab})^* \\ \Leftrightarrow & \mathtt{abab}x \in (\mathtt{ab})^+ \\ \Leftrightarrow & \mathtt{abab}x \in L \end{array}$$

In fact, the set of all strings with which `ab` is related turns out to be the set $\{(\texttt{ab})^n | n > 0\}$. This is usually referred to as the equivalence class of `ab` for $\equiv_L$, and written as $[\texttt{ab}]_{\equiv_L}$.

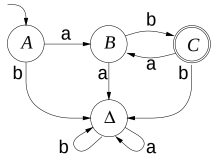
Similarly, we can show that $\texttt{a} \equiv_L \texttt{aba}$, and that $[\texttt{a}]_{\equiv_L} = (\texttt{ab})^*\texttt{a}$.

Using this kind of reasoning we end up with 4 distinct equivalent classes which span the whole of Σ^*. These are $[\varepsilon]_{\equiv_L}$, $[\texttt{a}]_{\equiv_L}$, $[\texttt{ab}]_{\equiv_L}$, $[\texttt{b}]_{\equiv_L}$.

Now consider the following DFSA, which we have already encountered before:



Note that all the strings in the equivalence class of `ab`, take the automaton from A to C. Similarly, all strings in the equivalence class of `b` take it from A to Δ and those in the equivalence class of ε take us from A to A. However, those in the equivalence class of `a`, take us from A either to B or to D. Can we construct a DFSA with the same language but where the equivalence classes of $\equiv_L$ are intimately related to single states? The answer in this case is positive, and we can construct the following automaton with the same language:



Intuitively, it should be clear that we cannot do any better than this and a DFSA with less than 4 states should be impossible. The following theorem formalizes and generalizes the arguments presented here and proves that such an automaton is in fact minimal and unique. This answers once and for all the questions we have been asking since the beginning of the chapter for regular languages.

Theorem: For every DFSA M there is a unique minimal DFSA M' such that M and M' are equivalent.

Proof: The proof is divided into steps as already discussed in section 7.2.1.

Part 1: We start by defining an equivalence relation based on the particular DFSA in question. Let $M = \langle K, T, t, k_1, F \rangle$. We define the relation over strings over alphabet T such that $x \equiv_M y$ if and only if both x and y take M from state k_1 to a particular state k':

$$\begin{array}{rcl} \equiv_M & \subseteq & \Sigma^* \times \Sigma^* \\ x \equiv_M y & \stackrel{def}{=} & t_*(k_1, x) = t_*(k_1, y) \end{array}$$

It is trivial to show that $\equiv_M$ has as many equivalence classes as there are states in M ($|K|$).

We now try to show that there is a lower limit on this number.

Part 2: We first prove that:

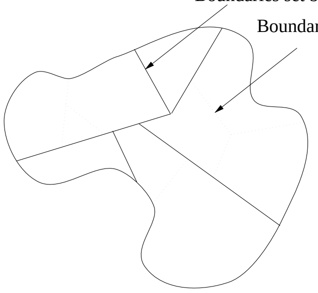
$$x \equiv_M y \Rightarrow x \equiv_L y$$

where L is the language recognized by M ($L = \mathcal{T}(M)$). The proof is rather straightforward:

$$\begin{array}{rl} & x \equiv_M y \\ \Rightarrow & \text{by definition of } \equiv_M \\ & t_*(k_1, x) = t_*(k_1, y) \\ \Rightarrow & \forall z : T^* \cdot t_*(t_*(k_1, x), z) = t_*(t_*(k_1, y), z) \\ \Rightarrow & \forall z : T^* \cdot t_*(k_1, xz) = t_*(k_1, yz) \\ \Rightarrow & \forall z : T^* \cdot t_*(k_1, xz) \in F \Leftrightarrow t_*(k_1, yz) \\ \Rightarrow & \text{by definition of } \mathcal{T}(M) \\ & \forall z : T^* \cdot xz \in \mathcal{T}(M) \Leftrightarrow yz \in \mathcal{T}(M) \\ \Rightarrow & \forall z : T^* \cdot xz \in L \Leftrightarrow yz \in L \\ \Rightarrow & x \equiv_L y \end{array}$$

Hence, it follows that for any string x, $[x]_{\equiv_M} \subseteq [x]_{\equiv_L}$.

This implies that no equivalence class of $\equiv_M$ may overlap over two or more equivalence classes of $\equiv_L$. Thus, the partitions created by $\equiv_L$ still hold when we consider those created by $\equiv_M$, except that new ones are created, as shown in the figure below.



This means that the number of equivalence classes of $\equiv_M$ cannot be smaller than the number of equivalence classes of $\equiv_L$. But the number of equivalence classes of $\equiv_M$ is the number of states of M:

$$|K| \geq |\{[x]_{\equiv_L} \mid x \in T^*\}|$$

Part 3: Now that we have set a lower bound on the number of states of M, can we construct an equivalent DFSA with this number of states? Consider the automaton $M_0 = \langle K', T, t', k'_1, F' \rangle$ where:

$$\begin{aligned}
K' &= \{[x]_{\equiv_L} \mid x \in T^*\} \\
t'([x]_{\equiv_L}, a) &= [xa]_{\equiv_L} \\
k'_1 &= [\varepsilon]_{\equiv_L} \\
F' &= \{[x]_{\equiv_L} \mid x \in L\}
\end{aligned}$$

Does this machine also generate language L?

$$\begin{aligned}
& \mathcal{T}(M_0) \\
= \; & \{x \mid t'_*(k'_1, x) \in F'\} \\
= \; & \{x \mid t'_*([\varepsilon]_{\equiv_L}, x) \in F'\} \\
= \; & \{x \mid [x]_{\equiv_L} \in F'\}
\end{aligned}$$

$$
\begin{aligned}
&= \{x \mid x \in L\} \\
&= L
\end{aligned}
$$

Part 4: We have thus proved that we can identify the minimum number of states of a DFSA which recognizes language L. Furthermore, we have identified one such automaton. The only thing left to prove is that it is unique.

How can we define uniqueness? Clearly, if we rename a state, the definitions would have changed, but without effectively changing the automaton in any relevant way. We thus define equality modulo state names of two automata M^1 and M^2 by proving that there is a mapping between the states of the machines such that:

- the mapping relates each state from M^1 with exactly one state in M^2 (and vice-versa).
- M^1 in state k goes to state k' upon input a if and only if M^2 in the state related to k can act upon input a to go to the state related to k'.

Assume that two DFSA M^1 and M^2 recognize language L, and both have exactly n states (where n is the number of equivalence classes of $\equiv_L$).

Let $M^i = \langle K^i, T, t^i, k_1^i, F^i \rangle$.

Clearly, by the reasoning in the previous steps, every state is related to one equivalent class of $\equiv_L$. Each state in K^1 is thus related to some $[x]_{\equiv_L}$. Similarly states in K^2.

The mapping between states is thus done via these equivalence classes. Now consider states $k^1 \in K^1$ and state $k^2 \in K^2$, where both are associated with $[x]_{\equiv_L}$.

Thus $t_*^i(k_1^i, x) = k^i$ for $i = 1$ or 2.

Now consider an input a:

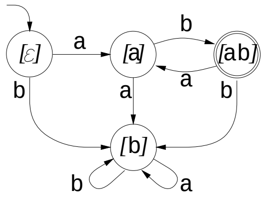
$$
\begin{aligned}
& t^1(k^1, a) \\
= \; & t^1(t^1_*(k^1_1, x), a) \\
= \; & t^1_*(k^1_1, xa)
\end{aligned}
$$

Thus, M^1 ends in the state related to $[xa]_{\equiv_L}$. Similar reasoning for M^2 gives precisely the same result. Hence, for any input received, it the automaton starts off in related states, it will also end in related states.

Hence, M^1 and M^2 are equal (modulo state naming).

$\square$

Example: The example we have been discussing can be renamed to:



7.2.3 Constructing a Minimal DFSA

The next problem is whether we can easily automate the process of minimizing a DFSA. The following procedure guarantees the minimization of an automaton to its simplest form:

1. Label the nodes using numbers 1, 2, $\ldots n$.
2. Construct matrices (of size $n \times n$) D^0, $D^1 \ldots$ using the following algorithm:

$$
D^0_{ij} = \begin{cases} \surd & \text{one of states } i, j \text{ is in } F \\ & \text{while the other is not} \\ \times & \text{otherwise} \end{cases}
$$

$$
D^{n+1}_{ij} = \begin{cases} \surd & \text{if } D^n_{ij} = T \text{ or if there is} \\ & \text{some } a \in T \text{ such that } D^n_{t(i,a)t(j,a)} = \surd \\ \times & \text{otherwise} \end{cases}
$$

3. Keep on constructing matrices until $D^r = D^{r+1}$.

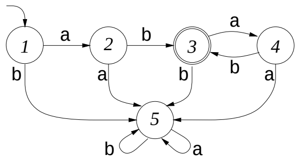
4. Now state i is indistinguishable from state j if and only if $D^r_{ij} = \surd$.

5. Join together indistinguishable states.

Note that a tick ($\surd$) in any of the matrices $D^n_{i,j}$ indicates that states i and j are distinguishable (different).

Why does this algorithm work? In the first step, when creating D_0, we say that two states are different if one is terminal while the other is not. In subsequent steps, we say that two states are different (either if we have already established so) or if, upon being given the same input, they evolve to states which have been shown to be different. The process is better understood by going through an example.

Example: Look at the following example (once again!):



First of all note that for any matrix D^i we construct, $D^i_{jk} = D^i_{kj}$. In other words, the matrix will be reflected across the main diagonal. To avoid confusion we will only fill in the top half of the matrix.

Construction of D^0: From the diagram, only state 3 is a final state. Thus all entries D^0_{13}, D^0_{23}, D^0_{34}, D^0_{35} are true (since 3 is in F, whereas 1, 2, 4 and 5 are not.

$$D^0 = \begin{pmatrix} \times & \times & \surd & \times & \times \\ & \times & \surd & \times & \times \\ & & \times & \surd & \surd \\ & & & \times & \times \\ & & & & \times \end{pmatrix}$$

Construction of D^1: We start off by copying all positions already set to $\surd$ (since they will remain so). By the definition of D^{n+1}, we notice that we will now distinguish nodes one of which evolves to 3 upon an input x, whereas the other evolves to a node which is not 3.

For example:

$$\begin{aligned} t(1, \mathtt{b}) &= 5 \\ t(2, \mathtt{b}) &= 3 \end{aligned}$$

Since $D^0_{53} = \surd$, D^1_{12} becomes $\surd$.

Similar reasons for changing states are given below:

$$\begin{aligned} t(1, \mathtt{b}) &= 5 \\ t(4, \mathtt{b}) &= 3 \\ &\text{hence } D^1_{41} \text{ becomes } \surd \\ t(2, \mathtt{b}) &= 3 \\ t(5, \mathtt{b}) &= 5 \\ &\text{hence } D^1_{52} \text{ becomes } \surd \\ t(4, \mathtt{b}) &= 3 \\ t(5, \mathtt{b}) &= 5 \\ &\text{hence } D^1_{54} \text{ becomes } \surd \end{aligned}$$

$$D^1 = \begin{pmatrix} \times & \surd & \surd & \surd & ? \\ & \times & \surd & ? & \surd \\ & & \times & \surd & \surd \\ & & & \times & \surd \\ & & & & \times \end{pmatrix}$$

Note that the diagonal entries must remain $\times$ since every node is always indistinguishable from itself. The algorithm also guarantees this. This leaves D^1_{51} and D^1_{42} undecided:

$$
\begin{aligned}
t(2,\mathtt{a}) &= 5 \\
t(4,\mathtt{a}) &= 5 \\
&\quad \text{where } D^0_{55} = \times \\
t(2,\mathtt{b}) &= 3 \\
t(4,\mathtt{b}) &= 3 \\
&\quad \text{where } D^0_{33} = \times
\end{aligned}
$$

Similarly:

$$
\begin{aligned}
t(1,\mathtt{a}) &= 2 \\
t(5,\mathtt{a}) &= 5 \\
&\quad \text{where } D^0_{25} = \times \\
t(1,\mathtt{b}) &= 5 \\
t(5,\mathtt{b}) &= 5 \\
&\quad \text{where } D^0_{55} = \times
\end{aligned}
$$

Hence both remain F:

$$
D^1 = \begin{pmatrix}
\times & \surd & \surd & \surd & \times \\
 & \times & \surd & \times & \surd \\
 & & \times & \surd & \surd \\
 & & & \times & \surd \\
 & & & & \times
\end{pmatrix}
$$

What about D^2? Again we copy all the $\surd$ entries and the main diagonal $\times$s. This time we note that:

$$
\begin{aligned}
t(1,\mathtt{a}) &= 2 \\
t(5,\mathtt{a}) &= 5 \\
&\quad \text{where } D^1_{25} = \surd
\end{aligned}
$$

and hence $D^2_{15} = \surd$. What about D^2_{24}?

$$\begin{array}{rcl}
t(2, \mathtt{a}) & = & 5 \\
t(4, \mathtt{a}) & = & 5 \\
 & & \text{where } D^1_{55} = \times \\
t(2, \mathtt{b}) & = & 3 \\
t(4, \mathtt{b}) & = & 3 \\
 & & \text{where } D^1_{33} = \times
\end{array}$$

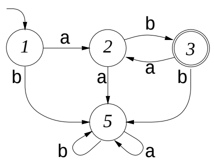
and hence remains $\times$.

$$D^2 \;=\; \begin{pmatrix} \times & \surd & \surd & \surd & \surd \\ & \times & \surd & \times & \surd \\ & & \times & \surd & \surd \\ & & & \times & \surd \\ & & & & \times \end{pmatrix}$$

Calculating D^3: There is now only D^3_{42} to consider. Again we get:

$$\begin{array}{rcl}
t(2, \mathtt{a}) & = & 5 \\
t(4, \mathtt{a}) & = & 5 \\
 & & \text{where } D^2_{55} = \times \\
t(2, \mathtt{b}) & = & 3 \\
t(4, \mathtt{b}) & = & 3 \\
 & & \text{where } D^2_{33} = \times
\end{array}$$

D^3 thus remains exactly like D^2, which allows us to stop generating matrices. What can we conclude from D^2? Apart from the main diagonal, only D^2_{24} is $\times$. This says that states 2 and 4 are indistinguishable and can thus be joined into a single state:



7.2.4 Exercises

Consider the following regular grammar:

$$
\begin{aligned}
G = \langle \Sigma, N, P, S \rangle & \\
\Sigma &= \{\mathtt{a},\ \mathtt{b}\} \\
N &= \{S,\ A,\ B\} \\
P &= \{\ S \to \mathtt{a}A \mid \mathtt{b} \\
&\qquad A \to \mathtt{a}B \\
&\qquad B \to \mathtt{a}A \mid \mathtt{b}\ \ \}
\end{aligned}
$$

1. Obtain a total DFSA M equivalent to G.
2. Minimize automaton M to get M_0.
3. Obtain a regular grammar G_0 equivalent M_0.

7.3 Context Free Grammars

The form for context free grammars is extremely general. A question arises naturally: Is there some way in which we can restrict the syntax of context free grammars without reducing their expressive power? The generality of context free grammars usually means more difficult proofs and inefficient parsing of the language. The aim of this section is to define two normal forms for context free grammars: the Chomsky normal form and Greibach normal form. The two different restrictions (on syntax) are aimed at different goals: the Chomsky normal form presents the grammar in a very restricted manner, making certain proofs about the language generated considerably easier. On the other hand, the Greibach normal form aims at producing a grammar for which the membership problem (is $x \in \mathcal{L}(G)$?) can be efficiently answered.

We note that neither of these normal forms is unique. In other words, non-equality of two grammars in either of these normal forms does not guarantee that the languages generated are different.

7.3.1 Chomsky Normal Form

Recall that productions allowed in context free grammars were of the form $A \to \alpha$, where A is a single non-terminal and α is a string of terminal and non-terminal symbols.

The Chomsky normal form allows only productions from a single non-terminal to either a single terminal symbol or to a pair of non-terminal symbols.

Definition: A grammar $G = \langle \Sigma, N, P, S \rangle$ is said to be in Chomsky normal form if all the production rules in P are of the form $A \to \alpha$, where $A \in N$ and $\alpha \in \Sigma \cup NN$.

Theorem: Any ε-free context free language L can be generated by a context free grammar in Chomsky normal form.

Construction: Let $G = \langle \Sigma, N, P, S \rangle$ by the ε-free context free grammar generating L.

Three steps are taken to progressively discard unwanted productions:

1. We start by getting rid of productions of the form $A \to B$, where $A, B \in N$.

2. We then replace all rules which produce strings which include terminal symbols (except for those which produce single terminal symbols).

3. Finally, we replace all rules which produce strings of more than two non-terminals.

Step 1: Consider rules of the form $A \to B$, where $A, B \in N$. We will replace all such rules with a common left hand side collectively.

If, for a non-terminal A, there is at least one production of the form $A \to B$, we replace all such rules by:

$$\{A \to \alpha \mid \exists C \in N \cdot C \to \alpha \in P,\ \alpha \notin N,\ A \stackrel{+}{\Rightarrow}_G C\}$$

In practice (see example) we would consider all productions which do not generate a single non-terminal, and check whether the left hand side can be derived from A.

Step 2: Consider rules of the form $A \rightarrow \alpha$, where α contains some terminal symbols (but $\alpha \notin \Sigma$). For each terminal symbol a, we replace all occurances of a by a new non-terminal T_a, and add the rule $T_a \rightarrow a$ to compensate.

Thus, rule $A \rightarrow aSba$ would be replaced by $A \rightarrow T_a S T_b T_a$ and the rules $T_a \rightarrow a$ and $T_b \rightarrow b$ are also added.

Step 3: Finally, we replace all rules which produce strings of (more than 2) non-terminal symbols. We replace rule $A \rightarrow B_1 B_2 \ldots B_n$ (where $A, B_i \in N$ and $n > 2$) with the family of rules:

$$\begin{array}{rcl} A & \rightarrow & B_1 B'_1 \\ B'_1 & \rightarrow & B_2 B'_2 \\ & \vdots & \\ B'_{n-2} & \rightarrow & B_{n-1} B_n \end{array}$$

Thus, $A \rightarrow ABCD$ would be replaced by:

$$\begin{array}{rcl} A & \rightarrow & AA'_1 \\ A'_1 & \rightarrow & BA'_2 \\ A'_2 & \rightarrow & CD \end{array}$$

It should be easy to see that this procedure actually produces a context free grammar in Chomsky normal form from any ε-free context free grammar. The fact that the new grammar generates the same language as the old one should also be intuitively obvious. The proof of this assertion can be found in the textbooks.

Example: Given the following grammar G, generate an equivalent context free grammar in Chomsky normal form.

$$G \stackrel{def}{=} \langle \{\mathtt{a}, \mathtt{b}\}, \{S, A, B\}, P, S \rangle$$

$$\begin{array}{rcll} P & \stackrel{def}{=} & \{ & S \rightarrow ASB \mid \mathtt{a} \\ & & & A \rightarrow B \mid \mathtt{b}B\mathtt{a} \\ & & & B \rightarrow S \mid \mathtt{a}S\mathtt{b} \quad \} \end{array}$$

Step 1: We start off by removing rules which produce a single non-terminal, of which we have $A \to B$ and $B \to S$.

- To eliminate all such productions with A on the left hand side (of which we only happen to have only one), we replace them by:

  $$\{A \to \alpha \mid \exists C \in N \cdot C \to \alpha \in P,\ \alpha \notin N,\ A \stackrel{+}{\Rightarrow}_G C\}$$

  We thus consider all rules going from non-terminal symbols derivable from A. Note that $A \stackrel{+}{\Rightarrow}_G S$, and $A \stackrel{+}{\Rightarrow}_G B$ (but not $A \stackrel{+}{\Rightarrow}_G A$):

  1. From $A \stackrel{+}{\Rightarrow}_G S$ and the production rules from S ($S \to ASB$ and $S \to \mathtt{a}$) we get:

     $$A \ \to \ ASB \mid \mathtt{a}$$

  2. Finally, from $A \stackrel{+}{\Rightarrow}_G B$ and $B \to \mathtt{a}S\mathtt{b}$, we get:

     $$A \ \to \ \mathtt{a}S\mathtt{b}$$

- To eliminate $B \to S$ (the only rule starting from B), we use the same procedure as for $A \to B$. Note that $B \stackrel{+}{\Rightarrow}_G S$ (but not A or B). Hence we only consider rules from S to get:

  $$B \ \to \ ASB \mid \mathtt{a}$$

Thus, at the end of the first step, we have:

$$\begin{array}{lcl} S & \to & ASB \mid \mathtt{a} \\ A & \to & ASB \mid \mathtt{a} \mid \mathtt{a}S\mathtt{b} \mid \mathtt{b}B\mathtt{a} \\ B & \to & ASB \mid \mathtt{a} \mid \mathtt{a}S\mathtt{b} \end{array}$$

Step 2: We now eliminate all rules producing α which includes terminal symbols (but $\alpha \notin \Sigma$).

Following the procedure described earlier, we add two new non-terminal symbols: T_a and T_b with related rules $T_a \to \texttt{a}$ and $T_b \to \texttt{b}$. We then replace all occurances of a and b appearing in the rules (in the form just described) with T_a and T_b respectively.

The resultant set of production rules is now:

$$\begin{aligned}
S &\to ASB \mid \texttt{a} \\
A &\to ASB \mid \texttt{a} \mid T_aST_b \mid T_bBT_a \\
B &\to ASB \mid \texttt{a} \mid T_aST_b \\
T_a &\to \texttt{a} \\
T_b &\to \texttt{b}
\end{aligned}$$

Step 3: Finally, we remove all rules which produce more that two non-terminal symbols, by progressively adding new non-terminals:

$$\begin{aligned}
S &\to AS' \mid \texttt{a} \\
S' &\to SB \\
A &\to AA' \mid \texttt{a} \mid T_aA'' \mid T_bA''' \\
A' &\to SB \\
A'' &\to ST_b \\
A''' &\to BT_a \\
B &\to AB' \mid \texttt{a} \mid T_aB'' \\
B' &\to SB \\
B'' &\to ST_b \\
T_a &\to \texttt{a} \\
T_b &\to \texttt{b}
\end{aligned}$$

This grammar is in Chomsky normal form.

Note that this is not unique. Clearly, we can remove redundant rules to obtain the more concise, but equivalent grammar also in Chomsky normal form:

$$
\begin{array}{rcl}
S & \rightarrow & AS' \mid \mathtt{a} \\
S' & \rightarrow & SB \\
A & \rightarrow & AS' \mid \mathtt{a} \mid T_a A' \mid T_b A'' \\
A' & \rightarrow & ST_b \\
A'' & \rightarrow & BT_a \\
B & \rightarrow & AS' \mid \mathtt{a} \mid T_a A' \\
T_a & \rightarrow & \mathtt{a} \\
T_b & \rightarrow & \mathtt{b}
\end{array}
$$

7.3.2 Greibach Normal Form

A grammar in Chomsky normal form may have a structure which makes it easier to prove properties about. But how about implementation of a parser for such a language. The new, less general form is still not very efficient to implement. How can we hope for a better, more efficient parsing?

Recall that one of the advantages of having a regular grammar associated with a language was that the right hand sides of the production rules started with a terminal symbol, thus enabling a more efficient parse of the language. Is such a normal form possible for general context free grammars? The Greibach normal form answers this positively. Let us start by defining exactly when we consider a grammar to be in Greibach normal form:

Definition: A grammar G is said to be in Greibach normal form if all the production rules are in the form $A \rightarrow a\alpha$, where $A \in N$, $a \in \Sigma$ and $\alpha \in (\Sigma \cup N)^*$.

As with the Chomsky normal form, for any context free grammar, we can construct an equivalent one in Greibach normal form. What is the approach taken?

First note that if any rule starts with a non-terminal, we can replace the non-terminal by the possible productions it could partake in. Thus, if $A \rightarrow B\alpha$ and $B \rightarrow \beta \mid \gamma$, we can replace the first rule by $A \rightarrow \beta\alpha \mid \gamma\alpha$. This process can be repeated until the rule starts with a terminal symbol. But would it always do so?

Consider $A \to a \mid Ab$. Clearly, no matter how many times we replace A, we will always get a production rule starting in a non-terminal. Productions of the form $A \to A\alpha$ are called left-recursive rules. We note that the rule given for A can generate strings a, ab, abb, etc. This can be generated by $A \to a \mid aA'$ and $A' \to b \mid bA'$. This can be generalized for more complex rule sets:

If $A \to A\alpha_1 \mid \ldots \mid A\alpha_n$ are all the left recursive productions from A, and $A \to \beta_1 \mid \ldots\beta_m$, are all the remaining productions from A, we can replace these production rules by:

$$
\begin{array}{rcl}
A & \to & \beta_1 \mid \ldots\beta_m \mid \beta_1 A' \mid \ldots\beta_m A' \\
A' & \to & \alpha_1 \mid \ldots\alpha_n \mid \alpha_1 A' \mid \ldots\alpha_n A'
\end{array}
$$

where A' is a new non-terminal symbol.

These two procedures will be used to construct the Greibach normal form of a general ε-free, context free grammar.

Theorem: For any ε-free context-free language L, there is a context free grammar in Greibach normal form which generated language L.

Construction: Let G be a context free grammar generating L. The procedure is then as follows:

Step 1: From G, produce an equivalent context free grammar in Chomsky normal form G'.

Step 2: Enumerate the states A_1 to A_n, such that S is renamed to A_1.

Step 3: For i starting from 1, increasing to n:

1. for any production $A_i \to A_j\alpha$, where $i > j$ perform the first procedure. Repeat this step while possible.
2. remove any left recursive productions from A_i using the second procedure (and introducing A_i').

At the end of this step, all production rules from A_i will produce a string starting either with a terminal symbol, or with a non-terminal A_j such that $j > i$. There will also be a number of rules from A'_i which start off with a terminal symbol or some A_j (*not* A'_j).

Step 4: For i, starting from n and going down to 1, if there is some production $A_i \to A_j\alpha$, repeatedly perform the first procedure.

This makes sure that all production rules from A_i are in the desired format. Note that since all rules from A'_i cannot start from a non-terminal A'_j, we can now easily complete the desired format:

Step 5: Replace $A'_i \to A_j\alpha$ by using the first procedure.

Example: Consider the following context free grammar in Chomsky normal form:

$$G \stackrel{def}{=} \langle\{\mathtt{a},\mathtt{b}\}, \{S, A, B\}, P, S\rangle$$

$$\begin{array}{rcll} P & \stackrel{def}{=} & \{ & S \to SA \mid BS \\ & & & A \to BB \mid \mathtt{a} \\ & & & B \to AA \mid \mathtt{b} \quad \} \end{array}$$

To produce an equivalent grammar in Greibach normal form, we start by enumerating the non-terminals, starting with S:

$$G' \stackrel{def}{=} \langle\{\mathtt{a},\mathtt{b}\}, \{A_1, A_2, A_3\}, P', A_1\rangle$$

$$\begin{array}{rcll} P' & \stackrel{def}{=} & \{ & A_1 \to A_1A_2 \mid A_3A_1 \\ & & & A_2 \to A_3A_3 \mid \mathtt{a} \\ & & & A_3 \to A_2A_2 \mid \mathtt{b} \quad \} \end{array}$$

We now perform step 3.

Consider $i = 1$. There are no productions of the form $A_1 \to A_j$, where $j < 1$, but there is one case of $A_1 \to A_1\alpha$. Using the second procedure to resolve it, we get:

$$\begin{array}{rcl} A_1 & \rightarrow & A_3A_1 \mid A_3A_1A_1' \\ A_1' & \rightarrow & A_2 \mid A_2A_1' \end{array}$$

When $i = 2$, we need to perform no changes:

$$A_2 \rightarrow A_3A_3 \mid \mathtt{a}$$

Finally, for $i = 3$, we first perform the first procedure on $A_3 \rightarrow A_2A_2$, obtaining:

$$A_3 \rightarrow A_3A_3A_2 \mid \mathtt{a}A_2 \mid \mathtt{b}$$

Since no more applications of the first procedure are possible, we now turn to the second.

$$\begin{array}{rcl} A_3 & \rightarrow & \mathtt{a}A_2A_3' \mid \mathtt{b}A_3' \mid \mathtt{a}A_2 \mid \mathtt{b} \\ A_3' & \rightarrow & A_3A_2 \mid A_3A_2A_3' \end{array}$$

Thus, at the end of step 3, we have:

$$\begin{array}{rcl} A_1 & \rightarrow & A_3A_1 \mid A_3A_1A_1' \\ A_1' & \rightarrow & A_2 \mid A_2A_1' \\ A_2 & \rightarrow & A_3A_3 \mid \mathtt{a} \\ A_3 & \rightarrow & \mathtt{a}A_2A_3' \mid \mathtt{b}A_3' \mid \mathtt{a}A_2 \mid \mathtt{b} \\ A_3' & \rightarrow & A_3A_2 \mid A_3A_2A_3' \end{array}$$

For step 4, we start with $i = 3$, where no modifications are necessary:

$$A_3 \rightarrow \mathtt{a}A_2A_3' \mid \mathtt{b}A_3' \mid \mathtt{a}A_2 \mid \mathtt{b}$$

With $i = 2$ we get:

$$A_2 \rightarrow \mathtt{b}A'_3A_3 \mid \mathtt{a}A_2A'_3A_3 \mid \mathtt{a} \mid \mathtt{a}A_2A_3 \mid \mathtt{b}A_3$$

Finally, with $i = 1$:

$$A_1 \rightarrow \mathtt{a}A_2A'_3A_1 \mid \mathtt{b}A'_3A_1 \mid \mathtt{a}A_2A'_3A_1A'_1 \mid \mathtt{b}A'_3A_1A'_1$$

Finally, we perform the first procedure on the new non-terminals A'_1 and A'_3:

$$\begin{aligned}
A'_1 &\rightarrow \mathtt{b}A'_3A_3 \mid \mathtt{a}A_2A'_3A_3 \mid \mathtt{a} \mid \mathtt{b}A'_3A_3A'_1 \mid \mathtt{a}A_2A'_3A_3A'_1 \mid \mathtt{a}A'_1 \\
A'_3 &\rightarrow \mathtt{a}A_2A'_3A_2 \mid \mathtt{b}A'_3A_2 \mid \mathtt{a}A_2A'_3A_2A'_3 \mid \mathtt{b}A'_3A_2A'_3
\end{aligned}$$

The production rules in the constructed Greibach normal form grammar are:

$$\begin{aligned}
A'_1 &\rightarrow \mathtt{b}A'_3A_3 \mid \mathtt{a}A_2A'_3A_3 \mid \mathtt{a} \mid \mathtt{b}A'_3A_3A'_1 \mid \mathtt{a}A_2A'_3A_3A'_1 \mid \mathtt{a}A'_1 \\
A'_3 &\rightarrow \mathtt{a}A_2A'_3A_2 \mid \mathtt{b}A'_3A_2 \mid \mathtt{a}A_2A'_3A_2A'_3 \mid \mathtt{b}A'_3A_2A'_3 \\
A_1 &\rightarrow \mathtt{a}A_2A'_3A_1 \mid \mathtt{b}A'_3A_1 \mid \mathtt{a}A_2A'_3A_1A'_1 \mid \mathtt{b}A'_3A_1A'_1 \\
A_2 &\rightarrow \mathtt{b}A'_3A_3 \mid \mathtt{a}A_2A'_3A_3 \mid \mathtt{a} \mid \mathtt{a}A_2A_3 \mid \mathtt{b}A_3 \\
A_3 &\rightarrow \mathtt{a}A_2A'_3 \mid \mathtt{b}A'_3 \mid \mathtt{a}A_2 \mid \mathtt{b}
\end{aligned}$$

As with the Chomsky normal form, the Greibach normal form is not unique. For example, we can add a new non-terminal symbol X with the related rules $\{X \rightarrow \alpha \mid A_1 \rightarrow \alpha\}$ and replace some instances of A_1 by X. Clearly, the two grammars are not identical even though they are equivalent and are both in Greibach normal form.

7.3.3 Exercises

1. For the following grammar, find an equivalent Chomsky normal form grammar:

$$G \stackrel{def}{=} \langle \{\mathtt{a}, \mathtt{b}\}, \{S\}, P, S \rangle$$
$$P \stackrel{def}{=} \{S \to \mathtt{a} \mid \mathtt{b} \mid \mathtt{a}S\mathtt{a} \mid \mathtt{b}S\mathtt{b}\}$$

2. Consider the ε-free context free grammar G:

$$G \stackrel{def}{=} \langle \{\mathtt{a}, \mathtt{b}\}, \{S, A, B\}, P, S \rangle$$
$$P \stackrel{def}{=} \left\{ \begin{array}{l} S \to ASB \mid \mathtt{a} \\ A \to AA \mid \mathtt{a} \\ B \to BB \mid \mathtt{b} \end{array} \right\}$$

   (a) Construct a grammar G' in Chomsky normal form, such that $\mathcal{L}(G) = \mathcal{L}(G')$.

   (b) Construct a grammar G'' in Greibach normal form, such that $\mathcal{L}(G) = \mathcal{L}(G'')$.

3. Show that any ε-free context free grammar can be transformed into an equivalent one, where all productions are of the form $A \to a\alpha$, such that $A \in N$, $a \in \Sigma$ and $\alpha \in N^*$.

   Produce such a grammar for the language given in question 2.

7.3.4 Conclusions

This section presented two possible normal forms for context free grammars. Despite not being unique normal forms, they are both useful for different reasons.

Note that we have not proved that the transformations presented do not change the language generated. Anybody interested in this should consult the textbooks.