## Contents

1. Introduction

2. Regular Languages, Minimization of DFA’s
   - 2.1 The Closure Definition of the Regular Languages
   - 2.2 Regular Expressions
   - 2.3 Regular Expressions and Regular Languages
   - 2.4 Regular Expressions and NFA’s
   - 2.5 Right-Invariant Equivalence Relations on $\Sigma^*$
   - 2.6 Finding minimal DFA’s
   - 2.7 State Equivalence and Minimal DFA’s
   - 2.8 The Pumping Lemma
   - 2.9 A Fast Algorithm for Checking State Equivalence

3. Context-Free Grammars And Languages
   - 3.1 Context-Free Grammars
   - 3.2 Derivations and Context-Free Languages
   - 3.3 Normal Forms for Context-Free Grammars
   - 3.4 Regular Languages are Context-Free
   - 3.5 Useless Productions in Context-Free Grammars
   - 3.6 The Greibach Normal Form
   - 3.7 Least Fixed-Points
   - 3.8 Context-Free Languages as Least Fixed-Points
   - 3.9 Least Fixed-Points and the Greibach Normal Form
   - 3.10 Tree Domains and Gorn Trees
   - 3.11 Derivations Trees
   - 3.12 Ogden’s Lemma

4. A Survey of $LR$-Parsing Methods
   - 4.1 $LR(0)$-Characteristic Automata
   - 4.2 Shift/Reduce Parsers
   - 4.3 Computation of FIRST
   - 4.4 The Intuition Behind the Shift/Reduce Algorithm
   - 4.5 The Graph Method for Computing Fixed Points
4.6 Computation of FOLLOW ........................................ 116
4.7 Algorithm Traverse ........................................... 117
4.8 More on LR(0)-Characteristic Automata ............................. 117
4.9 LALR(1)-Lookahead Sets ........................................ 119
4.10 Computing FIRST, FOLLOW, etc. in the Presence of ε-Rules .......... 121
4.11 LR(1)-Characteristic Automata ................................... 127

5 RAM Programs, Turing Machines .............................. 131

6 Universal RAM Programs and the Halting Problem ............... 133
6.1 Pairing Functions .............................................. 133
6.2 Equivalence of Alphabets ........................................ 136
6.3 Coding of RAM Programs ........................................ 139
6.4 Kleene’s T-Predicate ............................................ 147

7 Elementary Recursive Function Theory .......................... 151
7.1 Acceptable Indexings ............................................ 151
7.2 Undecidable Problems .......................................... 154
7.3 Recursively Enumerable Sets ..................................... 159
7.4 Reducibility and Complete Sets ................................ 164
7.5 The Recursion Theorem ......................................... 167
7.6 Extended Rice Theorem ......................................... 171
7.7 Creative and Productive Sets .................................... 174

8 The Post Correspondence Problem; Applications .................. 177
8.1 The Post Correspondence Problem ................................ 177
8.2 Some Undecidability Results for CFG’s .......................... 178
8.3 More Undecidability Properties of Languages ..................... 181

9 Computational Complexity; P and NP .......................... 183
9.1 The Class P ...................................................... 183
9.2 Directed Graphs, Paths ........................................ 185
9.3 Eulerian Cycles .................................................. 185
9.4 Hamiltonian Cycles ............................................. 187
9.5 Propositional Logic and Satisfiability .......................... 188
9.6 The Class NP, NP-Completeness ................................ 191
9.7 The Cook-Levin Theorem ....................................... 194

10 Some NP-Complete Problems ................................ 207
10.1 Statements of the Problems ..................................... 207
10.2 Proofs of NP-Completeness ..................................... 217
10.3 Succinct Certificates, coNP, and EXP .......................... 231
Chapter 1

Introduction

The theory of computation is concerned with algorithms and algorithmic systems: their design and representation, their completeness, and their complexity.

The purpose of these notes is to introduce some of the basic notions of the theory of computation, including concepts from formal languages and automata theory, the theory of computability, some basics of recursive function theory, and an introduction to complexity theory. Other topics such as correctness of programs will not be treated here (there just isn’t enough time!).

The notes are divided into three parts. The first part is devoted to formal languages and automata. The second part deals with models of computation, recursive functions, and undecidability. The third part deals with computational complexity, in particular the classes $\mathcal{P}$ and $\mathcal{NP}$. 
Chapter 2

Regular Languages and Equivalence Relations, The Myhill-Nerode Characterization, State Equivalence

2.1 The Closure Definition of the Regular Languages

Let $\Sigma = \{a_1, \ldots, a_m\}$ be some alphabet. We would like to define a family of languages, $R(\Sigma)$, by singling out some very basic (atomic) languages, namely the languages $\{a_1\}, \ldots, \{a_m\}$, the empty language, and the trivial language, $\{\epsilon\}$, and then forming more complicated languages by repeatedly forming union, concatenation and Kleene $^*$ of previously constructed languages. By doing so, we hope to get a family of languages ($R(\Sigma)$) that is closed under union, concatenation, and Kleene $^*$. This means that for any two languages, $L_1, L_2 \in R(\Sigma)$, we also have $L_1 \cup L_2 \in R(\Sigma)$ and $L_1 L_2 \in R(\Sigma)$, and for any language $L \in R(\Sigma)$, we have $L^* \in R(\Sigma)$. Furthermore, we would like $R(\Sigma)$ to be the smallest family with these properties. How do we achieve this rigorously?

First, let us look more closely at what we mean by a family of languages. Recall that a language (over $\Sigma$) is any subset, $L$, of $\Sigma^*$. Thus, the set of all languages is $2^{\Sigma^*}$, the power set of $\Sigma^*$. If $\Sigma$ is nonempty, this is an uncountable set. Next, we define a family, $\mathcal{L}$, of languages to be any subset of $2^{\Sigma^*}$. This time, the set of families of languages is $2^{2^{\Sigma^*}}$. This is a huge set. We can use the inclusion relation on $2^{2^{\Sigma^*}}$ to define a partial order on families of languages. So, $\mathcal{L}_1 \subseteq \mathcal{L}_2$ iff for every language, $L$, if $L \in \mathcal{L}_1$ then $L \in \mathcal{L}_2$.

We can now state more precisely what we are trying to do. Consider the following properties for a family of languages, $\mathcal{L}$:

1. We have $\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\} \in \mathcal{L}$, i.e., $\mathcal{L}$ contains the “atomic” languages.

2a. For all $L_1, L_2 \in \mathcal{L}$, we also have $L_1 \cup L_2 \in \mathcal{L}$.

2b. For all $L_1, L_2 \in \mathcal{L}$, we also have $L_1 L_2 \in \mathcal{L}$. 

7
(2c) For all \( L \in \mathcal{L} \), we also have \( L^* \in \mathcal{L} \).

In other words, \( \mathcal{L} \) is closed under union, concatenation and Kleene *.

Now, what we want is the smallest (w.r.t. inclusion) family of languages that satisfies properties (1) and (2)(a)(b)(c). We can construct such a family using an inductive definition. This inductive definition constructs a sequence of families of languages, \((R(\Sigma)_n)_{n \geq 0}\), called the stages of the inductive definition, as follows:

\[
R(\Sigma)_0 = \{\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\}\},
\]

\[
R(\Sigma)_{n+1} = R(\Sigma)_n \cup \{L_1 \cup L_2, L_1L_2, L^* \mid L_1, L_2, L \in R(\Sigma)_n\}.
\]

Then, we define \( R(\Sigma) \) by

\[
R(\Sigma) = \bigcup_{n \geq 0} R(\Sigma)_n.
\]

Thus, a language \( L \) belongs to \( R(\Sigma) \) iff it belongs \( L_n \), for some \( n \geq 0 \).

For example, if \( \Sigma = \{a, b\} \), we have

\[
R(\Sigma)_1 = \{\{a\}, \{b\}, \emptyset, \{\epsilon\}, \{a, b\}, \{a, \epsilon\}, \{b, \epsilon\}, \{ab\}, \{ba\}, \{aa\}, \{bb\}, \{a\}^*, \{b\}^*\}.
\]

Some of the languages that will appear in \( R(\Sigma)_2 \) are:

\[
\{a, bb\}, \{ab, ba\}, \{abb\}, \{aabb\}, \{a\} \{a\}^*, \{aa\} \{b\}^*, \{bb\}^*.
\]

Observe that

\[
R(\Sigma)_0 \subseteq R(\Sigma)_1 \subseteq R(\Sigma)_2 \subseteq \cdots \subseteq R(\Sigma)_n \subseteq R(\Sigma)_{n+1} \subseteq \cdots \subseteq R(\Sigma),
\]

so that if \( L \in R(\Sigma)_n \), then \( L \in R(\Sigma)_p \), for all \( p \geq n \). Also, there is some smallest \( n \) for which \( L \in R(\Sigma)_n \) (the birthdate of \( L \)). In fact, all these inclusions are strict. Note that each \( R(\Sigma)_n \) only contains a finite number of languages (but some of the languages in \( R(\Sigma)_n \) are infinite, because of Kleene *).

Then we define the Regular languages, Version 2, as the family \( R(\Sigma) \).

Of course, it is far from obvious that \( R(\Sigma) \) coincides with the family of languages accepted by DFA’s (or NFA’s), what we call the regular languages, version 1. However, this is the case, and this can be demonstrated by giving two algorithms. Actually, it will be slightly more convenient to define a notation system, the regular expressions, to denote the languages in \( R(\Sigma) \). Then, we will give an algorithm that converts a regular expression, \( R \), into an NFA, \( N_R \), so that \( L_R = L(N_R) \), where \( L_R \) is the language (in \( R(\Sigma) \)) denoted by \( R \). We will also give an algorithm that converts an NFA, \( N \), into a regular expression, \( R_N \), so that \( L(R_N) = L(N) \).

But before doing all this, we should make sure that \( R(\Sigma) \) is indeed the family that we are seeking. This is the content of
Proposition 2.1. The family, \( R(\Sigma) \), is the smallest family of languages which contains the atomic languages \( \{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\} \), and is closed under union, concatenation, and Kleene *.

Proof. There are two things to prove.

(i) We need to prove that \( R(\Sigma) \) has properties (1) and (2)(a)(b)(c).

(ii) We need to prove that \( R(\Sigma) \) is the smallest family having properties (1) and (2)(a)(b)(c).

(i) Since

\[
R(\Sigma)_0 = \{\{a_1\}, \ldots, \{a_m\}, \emptyset, \{\epsilon\}\},
\]

it is obvious that (1) holds. Next, assume that \( L_1, L_2 \in R(\Sigma) \). This means that there are some integers \( n_1, n_2 \geq 0 \), so that \( L_1 \in R(\Sigma)_{n_1} \) and \( L_2 \in R(\Sigma)_{n_2} \). Now, it is possible that \( n_1 \neq n_2 \), but if we let \( n = \max\{n_1, n_2\} \), as we observed that \( R(\Sigma)_p \subseteq R(\Sigma)_q \) whenever \( p \leq q \), we are guaranteed that both \( L_1, L_2 \in R(\Sigma)_n \). However, by the definition of \( R(\Sigma)_{n+1} \) (that’s why we defined it this way!), we have \( L_1 \cup L_2 \in R(\Sigma)_{n+1} \subseteq R(\Sigma) \). The same argument proves that \( L_1 L_2 \in R(\Sigma)_{n+1} \subseteq R(\Sigma) \). Also, if \( L \in R(\Sigma)_n \), we immediately have \( L^* \in R(\Sigma)_{n+1} \subseteq R(\Sigma) \). Therefore, \( R(\Sigma) \) has properties (1) and (2)(a)(b)(c).

(ii) Let \( \mathcal{L} \) be any family of languages having properties (1) and (2)(a)(b)(c). We need to prove that \( R(\Sigma) \subseteq \mathcal{L} \). If we can prove that \( R(\Sigma)_n \subseteq \mathcal{L} \), for all \( n \geq 0 \), we are done (since then, \( R(\Sigma) = \bigcup_{n \geq 0} R(\Sigma)_n \subseteq \mathcal{L} \)). We prove by induction on \( n \) that \( R(\Sigma)_n \subseteq \mathcal{L} \), for all \( n \geq 0 \).

The base case \( n = 0 \) is trivial, since \( \mathcal{L} \) has (1), which says that \( R(\Sigma)_0 \subseteq \mathcal{L} \). Assume inductively that \( R(\Sigma)_n \subseteq \mathcal{L} \). We need to prove that \( R(\Sigma)_{n+1} \subseteq \mathcal{L} \). Pick any \( L \in R(\Sigma)_{n+1} \). Recall that

\[
R(\Sigma)_{n+1} = R(\Sigma)_n \cup \{L_1 \cup L_2, L_1 L_2, L^* \mid L_1, L_2, L \in R(\Sigma)_n\}.
\]

If \( L \in R(\Sigma)_n \), then \( L \in \mathcal{L} \), since \( R(\Sigma)_n \subseteq \mathcal{L} \), by the induction hypothesis. Otherwise, there are three cases:

(a) \( L = L_1 \cup L_2 \), where \( L_1, L_2 \in R(\Sigma)_n \). By the induction hypothesis, \( R(\Sigma)_n \subseteq \mathcal{L} \), so, we get \( L_1, L_2 \in \mathcal{L} \); since \( \mathcal{L} \) has 2(a), we have \( L_1 \cup L_2 \in \mathcal{L} \).

(b) \( L = L_1 L_2 \), where \( L_1, L_2 \in R(\Sigma)_n \). By the induction hypothesis, \( R(\Sigma)_n \subseteq \mathcal{L} \), so, we get \( L_1, L_2 \in \mathcal{L} \); since \( \mathcal{L} \) has 2(b), we have \( L_1 L_2 \in \mathcal{L} \).

(c) \( L = L_1^* \), where \( L_1 \in R(\Sigma)_n \). By the induction hypothesis, \( R(\Sigma)_n \subseteq \mathcal{L} \), so, we get \( L_1 \in \mathcal{L} \); since \( \mathcal{L} \) has 2(c), we have \( L_1^* \in \mathcal{L} \).

Thus, in all cases, we showed that \( L \in \mathcal{L} \), and so, \( R(\Sigma)_{n+1} \subseteq \mathcal{L} \), which proves the induction step. \( \square \)
Note: a given language $L$ may be built up in different ways. For example,

$$\{a, b\}^* = (\{a\}^* \{b\}^*)^*.$$

Students should study carefully the above proof. Although simple, it is the prototype of many proofs appearing in the theory of computation.

### 2.2 Regular Expressions

The definition of the family of languages $R(\Sigma)$ given in the previous section in terms of an inductive definition is good to prove properties of these languages but is not very convenient to manipulate them in a practical way. To do so, it is better to introduce a symbolic notation system, the *regular expressions*. Regular expressions are certain strings formed according to rules that mimic the inductive rules for constructing the families $R(\Sigma)_n$.

The set of regular expressions $R(\Sigma)$ over an alphabet $\Sigma$ is a language defined on an alphabet $\Delta$ defined as follows.

Given an alphabet $\Sigma = \{a_1, \ldots, a_m\}$, consider the new alphabet

$$\Delta = \Sigma \cup \{+,*,(,),\emptyset,\epsilon\}.$$

We define the family $(R(\Sigma)_n)$ of languages over $\Delta$ as follows:

$$R(\Sigma)_0 = \{a_1, \ldots, a_m, \emptyset, \epsilon\},$$

$$R(\Sigma)_{n+1} = R(\Sigma)_n \cup \{(R_1 + R_2), (R_1 \cdot R_2), R^* | R_1, R_2, R \in R(\Sigma)_n\}.$$

Then, we define $R(\Sigma)$ as

$$R(\Sigma) = \bigcup_{n \geq 0} R(\Sigma)_n.$$  

Note that every language $R(\Sigma)_n$ is finite.

For example, if $\Sigma = \{a, b\}$, we have

$$R(\Sigma)_1 = \{a, b, \emptyset, \epsilon, (a + b), (b + a), (a + a), (b + b), (a + \epsilon), (\epsilon + a), (b + \epsilon), (\epsilon + b), (a + \emptyset), (\emptyset + a), (b + \emptyset), (\emptyset + b), (\epsilon + \epsilon), (\epsilon + \emptyset), (\emptyset + \epsilon), (\emptyset + \emptyset), (a \cdot b), (b \cdot a), (a \cdot a), (b \cdot b), (a \cdot \epsilon), (\epsilon \cdot a), (b \cdot \epsilon), (\epsilon \cdot b), (\epsilon \cdot \epsilon), (a \cdot \emptyset), (\emptyset \cdot a), (b \cdot \emptyset), (\emptyset \cdot b), (\epsilon \cdot \emptyset), (\emptyset \cdot \epsilon), (\emptyset \cdot \emptyset), a^*, b^*, \epsilon^*, \emptyset^*\}.$$
2.3. REGULAR EXPRESSIONS AND REGULAR LANGUAGES

Some of the regular expressions appearing in \( \mathcal{R}(\Sigma)_2 \) are:

\[
(a + (b \cdot b)), ((a \cdot b) + (b \cdot a)), ((a \cdot b) \cdot b),
\]

\[
((a \cdot a) \cdot (b \cdot b)), (a \cdot a^*), ((a \cdot a) \cdot b^*), (b \cdot b)^*.
\]

**Definition 2.1.** The set \( \mathcal{R}(\Sigma) \) is the set of regular expressions (over \( \Sigma \)).

**Proposition 2.2.** The language \( \mathcal{R}(\Sigma) \) is the smallest language which contains the symbols \( a_1, \ldots, a_m, \emptyset, \epsilon \), from \( \Delta \), and such that \( (R_1 + R_2) \), \( (R_1 \cdot R_2) \), and \( R^* \), also belong to \( \mathcal{R}(\Sigma) \), when \( R_1, R_2, R \in \mathcal{R}(\Sigma) \).

For simplicity of notation, we write

\[
(R_1R_2)
\]

instead of

\[
(R_1 \cdot R_2).
\]

**Examples:** \( R = (a + b)^* \), \( S = (a^*b^*)^* \).

\[
T = ((a + b)^*a)((a + b) \cdots (a + b)).
\]

2.3 Regular Expressions and Regular Languages

Every regular expression \( R \in \mathcal{R}(\Sigma) \) can be viewed as the name, or denotation, of some language \( L \in \mathcal{R}(\Sigma) \). Similarly, every language \( L \in \mathcal{R}(\Sigma) \) is the interpretation (or meaning) of some regular expression \( R \in \mathcal{R}(\Sigma) \).

Think of a regular expression \( R \) as a program, and of \( \mathcal{L}(R) \) as the result of the execution or evaluation, of \( R \) by \( \mathcal{L} \).

This can be made rigorous by defining a function

\[
\mathcal{L}: \mathcal{R}(\Sigma) \rightarrow \mathcal{R}(\Sigma).
\]

This function is defined recursively as follows:

\[
\mathcal{L}[a_i] = \{a_i\},
\]

\[
\mathcal{L}[\emptyset] = \emptyset,
\]

\[
\mathcal{L}[\epsilon] = \{\epsilon\},
\]

\[
\mathcal{L}[(R_1 + R_2)] = \mathcal{L}[R_1] \cup \mathcal{L}[R_2],
\]

\[
\mathcal{L}[(R_1 \cdot R_2)] = \mathcal{L}[R_1] \mathcal{L}[R_2],
\]

\[
\mathcal{L}[R^*] = \mathcal{L}[R]^*.
\]
Proposition 2.3. For every regular expression \( R \in \mathcal{R}(\Sigma) \), the language \( \mathcal{L}[R] \) is regular (version 2), i.e. \( \mathcal{L}[R] \in \mathcal{R}(\Sigma) \). Conversely, for every regular (version 2) language \( L \in \mathcal{R}(\Sigma) \), there is some regular expression \( R \in \mathcal{R}(\Sigma) \) such that \( L = \mathcal{L}[R] \).

Proof. To prove that \( \mathcal{L}[R] \in \mathcal{R}(\Sigma) \) for all \( R \in \mathcal{R}(\Sigma) \), we prove by induction on \( n \geq 0 \) that if \( R \in \mathcal{R}(\Sigma)^n \), then \( \mathcal{L}[R] \in \mathcal{R}(\Sigma)^n \). To prove that \( L \) is surjective, we prove by induction on \( n \geq 0 \) that if \( L \in \mathcal{R}(\Sigma)^n \), then there is some \( R \in \mathcal{R}(\Sigma)^n \) such that \( L = \mathcal{L}[R] \).

Note: the function \( \mathcal{L} \) is not injective.

Example: If \( R = (a + b)^* \), \( S = ((ab)^*a)^* \), then
\[
\mathcal{L}[R] = \mathcal{L}[S] = \{a, b\}^*.
\]

For simplicity, we often denote \( \mathcal{L}[R] \) as \( L_R \). As examples, we have
\[
\begin{align*}
\mathcal{L}[(ab + ab)] &= \{a, abb\} \\
\mathcal{L}[(ab^*a^*b)^*a^*] &= \{w \in \{a, b\}^* \mid w \text{ has two } b's\} \\
\mathcal{L}[(ab^*a^*b)a^*] &= \{w \in \{a, b\}^* \mid w \text{ has an even } \# \text{ of } b's\} \\
\mathcal{L}[a^*] &= \{w \in \{a, b\}^* \mid w \text{ has an odd } \# \text{ of } b's\}
\end{align*}
\]

Remark. If
\[
R = ((a + b)^*a)((a + b)\cdots(a + b))_n,
\]

it can be shown that any minimal DFA accepting \( L_R \) has \( 2^{n+1} \) states. Yet, both \( ((a + b)^*a \) and \( ((a + b)\cdots(a + b))_n \) denote languages that can be accepted by “small” DFA’s (of size 2 and \( n + 2 \)).

Definition 2.2. Two regular expressions \( R, S \in \mathcal{R}(\Sigma) \) are equivalent, denoted as \( R \cong S \), iff \( \mathcal{L}[R] = \mathcal{L}[S] \).

It is immediate that \( \cong \) is an equivalence relation. The relation \( \cong \) satisfies some (nice) identities. For example:
\[
\begin{align*}
((aa) + b) + c &\cong ((aa) + (b + c)) \\
(aa)(bc) &\cong ((aa)b)(cc) \\
(a^*a^*) &\cong a^*.
\end{align*}
\]
and more generally

\[
((R_1 + R_2) + R_3) \equiv (R_1 + (R_2 + R_3)),
\]
\[
((R_1R_2)R_3) \equiv (R_1(R_2R_3)),
\]
\[
(R_1 + R_2) \equiv (R_2 + R_1),
\]
\[
(R^* R^*) \equiv R^*,
\]
\[
R^{**} \equiv R^*.
\]

There is an algorithm to test the equivalence of regular expressions, but its complexity is exponential. Such an algorithm uses the conversion of a regular expression to an NFA, and the subset construction for converting an NFA to a DFA. Then the problem of deciding whether two regular expressions \( R \) and \( S \) are equivalent is reduced to testing whether two DFA \( D_1 \) and \( D_2 \) accept the same languages (the equivalence problem for DFA’s). This last problem is equivalent to testing whether

\[
L(D_1) - L(D_2) = \emptyset \text{ and } L(D_2) - L(D_1) = \emptyset.
\]

But \( L(D_1) - L(D_2) \) (and similarly \( L(D_2) - L(D_1) \)) is accepted by a DFA obtained by the cross-product construction for the relative complement (with final states \( F_1 \times \overline{F_2} \) and \( \overline{F_1} \times F_2 \)). Thus, in the end, the equivalence problem for regular expressions reduces to the problem of testing whether a DFA \( D = (Q, \Sigma, \delta, q_0, F) \) accepts the empty language, which is equivalent to \( Q \cap F = \emptyset \). This last problem is a reachability problem in a directed graph which is easily solved in polynomial time.

It is an open problem to prove that the problem of testing the equivalence of regular expressions cannot be decided in polynomial time.

In the next two sections we show the equivalence of NFA’s and regular expressions, by providing an algorithm to construct an NFA from a regular expression, and an algorithm for constructing a regular expression from an NFA. This will show that the regular languages Version 1 coincide with the regular languages Version 2.

### 2.4 Regular Expressions and NFA’s

**Proposition 2.4.** There is an algorithm, which, given any regular expression \( R \in \mathcal{R}(\Sigma) \), constructs an NFA \( N_R \) accepting \( L_R \), i.e., such that \( L_R = L(N_R) \).

In order to ensure the correctness of the construction as well as to simplify the description of the algorithm it is convenient to assume that our NFA’s satisfy the following conditions:

1. Each NFA has a single final state, \( t \), distinct from the start state, \( s \).
2. There are no incoming transitions into the the start state, \( s \), and no outgoing transitions from the final state, \( t \).
3. Every state has at most two incoming and two outgoing transitions.

Here is the algorithm.

For the base case, either

(a) $R = a_i$, in which case, $N_R$ is the following NFA:

$$
\begin{array}{c}
  s \xrightarrow{a_i} t \\
\end{array}
$$

Figure 2.1: NFA for $a_i$

(b) $R = \epsilon$, in which case, $N_R$ is the following NFA:

$$
\begin{array}{c}
  s \xrightarrow{\epsilon} t \\
\end{array}
$$

Figure 2.2: NFA for $\epsilon$

(c) $R = \emptyset$, in which case, $N_R$ is the following NFA:

$$
\begin{array}{c}
  s \quad t \\
\end{array}
$$

Figure 2.3: NFA for $\emptyset$

The recursive clauses are as follows:

(i) If our expression is $(R+S)$, the algorithm is applied recursively to $R$ and $S$, generating NFA’s $N_R$ and $N_S$, and then these two NFA’s are combined in parallel as shown in Figure 2.4:

$$
\begin{array}{c}
  s \xrightarrow{\epsilon} s_1 \xrightarrow{\epsilon} t_1 \\
  s \xrightarrow{\epsilon} s_2 \xrightarrow{\epsilon} t_2
\end{array}
$$

Figure 2.4: NFA for $(R+S)$
(ii) If our expression is \((R \cdot S)\), the algorithm is applied recursively to \(R\) and \(S\), generating NFA’s \(N_R\) and \(N_S\), and then these NFA’s are combined sequentially as shown in Figure 2.5 by merging the “old” final state, \(t_1\), of \(N_R\), with the “old” start state, \(s_2\), of \(N_S\):

![Figure 2.5: NFA for \((R \cdot S)\)](image)

Note that since there are no incoming transitions into \(s_2\) in \(N_S\), once we enter \(N_S\), there is no way of reentering \(N_R\), and so the construction is correct (it yields the concatenation \(L_RL_S\)).

(iii) If our expression is \(R^*\), the algorithm is applied recursively to \(R\), generating the NFA \(N_R\). Then we construct the NFA shown in Figure 2.6 by adding an \(\epsilon\)-transition from the “old” final state, \(t_1\), of \(N_R\) to the “old” start state, \(s_1\), of \(N_R\) and, as \(\epsilon\) is not necessarily accepted by \(N_R\), we add an \(\epsilon\)-transition from \(s\) to \(t\):

![Figure 2.6: NFA for \(R^*\)](image)

Since there are no outgoing transitions from \(t_1\) in \(N_R\), we can only loop back to \(s_1\) from \(t_1\) using the new \(\epsilon\)-transition from \(t_1\) to \(s_1\) and so the NFA of Figure 2.6 does accept \(N_R^*\).

The algorithm that we just described is sometimes called the “sombrero construction.”

As a corollary of this construction, we get

Reg. languages version 2 \(\subseteq\) Reg. languages, version 1.

The reader should check that if one constructs the NFA corresponding to the regular expression \((a + b)^*abb\) and then applies the subset construction, one get the following DFA:
We now consider the construction of a regular expression from an NFA.

**Proposition 2.5.** There is an algorithm, which, given any NFA $N$, constructs a regular expression $R \in \mathcal{R}(\Sigma)$, denoting $L(N)$, i.e., such that $L_R = L(N)$.

As a corollary,

Reg. languages version 1 $\subseteq$ Reg. languages, version 2.

This is the **node elimination algorithm**.

The general idea is to allow more general labels on the edges of an NFA, namely, regular expressions. Then, such generalized NFA’s are simplified by eliminating nodes one at a time, and readjusting labels.

**Preprocessing, phase 1:**

If necessary, we need to add a new start state with an $\epsilon$-transition to the old start state, if there are incoming edges into the old start state.

If necessary, we need to add a new (unique) final state with $\epsilon$-transitions from each of the old final states to the new final state, if there is more than one final state or some outgoing edge from any of the old final states.

At the end of this phase, the start state, say $s$, is a source (no incoming edges), and the final state, say $t$, is a sink (no outgoing edges).

**Preprocessing, phase 2:**

We need to “flatten” parallel edges. For any pair of states $(p, q)$ ($p = q$ is possible), if there are $k$ edges from $p$ to $q$ labeled $u_1, \ldots, u_k$, then create a single edge labeled with the regular expression

$$u_1 + \cdots + u_k.$$
For any pair of states \((p, q)\) \((p = q\) is possible\) such that there is no edge from \(p\) to \(q\), we put an edge labeled \(\emptyset\).

At the end of this phase, the resulting "generalized NFA" is such that for any pair of states \((p, q)\) (where \(p = q\) is possible), there is a unique edge labeled with some regular expression denoted as \(R_{p,q}\). When \(R_{p,q} = \emptyset\), this really means that there is no edge from \(p\) to \(q\) in the original NFA \(N\).

By interpreting each \(R_{p,q}\) as a function call (really, a macro) to the NFA \(N_{p,q}\) accepting \(L[R_{p,q}]\) (constructed using the previous algorithm), we can verify that the original language \(L(N)\) is accepted by this new generalized NFA.

**Node elimination** only applies if the generalized NFA has at least one node distinct from \(s\) and \(t\).

Pick any node \(r\) distinct from \(s\) and \(t\). For every pair \((p, q)\) where \(p \neq r\) and \(q \neq r\), replace the label of the edge from \(p\) to \(q\) as indicated below:

![Figure 2.8: Before Eliminating node \(r\)](image-url)
At the end of this step, delete the node $r$ and all edges adjacent to $r$.

Note that $p = q$ is possible, in which case the triangle is “flat”. It is also possible that $p = s$ or $q = t$. Also, this step is performed for all pairs $(p, q)$, which means that both $(p, q)$ and $(q, p)$ are considered (when $p \neq q$).

Note that this step only has an effect if there are edges from $p$ to $r$ and from $r$ to $q$ in the original NFA $N$. Otherwise, $r$ can simply be deleted, as well as the edges adjacent to $r$.

Other simplifications can be made. For example, when $R_{r,r} = \emptyset$, we can simplify $R_{p,r} R_{r,r} R_{r,q}$ to $R_{p,r} R_{r,q}$. When $R_{p,q} = \emptyset$, we have $R_{p,r} R_{r,r} R_{r,q}$.

The order in which the nodes are eliminated is irrelevant, although it affects the size of the final expression.

The algorithm stops when the only remaining nodes are $s$ and $t$. Then, the label $R$ of the edge from $s$ to $t$ is a regular expression denoting $L(N)$.

For example, let

$$L = \{ w \in \Sigma^* \mid w \text{ contains an odd number of } a\text{'s or an odd number of } b\text{'s} \}.$$ 

An NFA for $L$ after the preprocessing phase is:
After eliminating node 2:

After eliminating node 3:
After eliminating node 4:

\[
T = a + b + (ab + ba)(aa + bb)^*(\epsilon + a + b)
\]

and

\[
S = aa + bb + (ab + ba)(aa + bb)^*(ab + ba).
\]

Finally, after eliminating node 1, we get:

\[
R = (aa + bb + (ab + ba)(aa + bb)^*(ab + ba))\times(a + b + (ab + ba)(aa + bb)^*(\epsilon + a + b)).
\]
2.5 Right-Invariant Equivalence Relations on $\Sigma^*$

The purpose of this section is to give one more characterization of the regular languages in terms of certain kinds of equivalence relations on strings. Pushing this characterization a bit further, we will be able to show how minimal DFA’s can be found.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The DFA $D$ may be redundant, for example, if there are states that are not accessible from the start state. The set $Q_r$ of accessible or reachable states is the subset of $Q$ defined as

$$Q_r = \{ p \in Q \mid \exists w \in \Sigma^*, \delta^*(q_0, w) = p \}.$$ 

If $Q \neq Q_r$, we can “clean up” $D$ by deleting the states in $Q - Q_r$ and restricting the transition function $\delta$ to $Q_r$. This way, we get an equivalent DFA $D_r$ such that $L(D) = L(D_r)$, where all the states of $D_r$ are reachable. From now on, we assume that we are dealing with DFA’s such that $D = D_r$, called trim, or reachable.

Recall that an equivalence relation $\simeq$ on a set $A$ is a relation which is reflexive, symmetric, and transitive. Given any $a \in A$, the set

$$\{ b \in A \mid a \simeq b \}$$

is called the equivalence class of $a$, and it is denoted as $[a]$ or even as $[a]$. Recall that for any two elements $a, b \in A$, $[a] \cap [b] = \emptyset$ iff $a \not\simeq b$, and $[a] = [b]$ iff $a \simeq b$. The set of equivalence classes associated with the equivalence relation $\simeq$ is a partition $\Pi$ of $A$ (also denoted as $A/\simeq$). This means that it is a family of nonempty pairwise disjoint sets whose union is equal to $A$ itself. The equivalence classes are also called the blocks of the partition $\Pi$. The number of blocks in the partition $\Pi$ is called the index of $\simeq$ (and $\Pi$).

Given any two equivalence relations $\simeq_1$ and $\simeq_2$ with associated partitions $\Pi_1$ and $\Pi_2$,

$$\simeq_1 \subseteq \simeq_2$$

iff every block of the partition $\Pi_1$ is contained in some block of the partition $\Pi_2$. Then, every block of the partition $\Pi_2$ is the union of blocks of the partition $\Pi_1$, and we say that $\simeq_1$ is a refinement of $\simeq_2$ (and similarly, $\Pi_1$ is a refinement of $\Pi_2$). Note that $\Pi_2$ has at most as many blocks as $\Pi_1$ does.

We now define an equivalence relation on strings induced by a DFA. This equivalence is a kind of “observational” equivalence, in the sense that we decide that two strings $u, v$ are equivalent iff, when feeding first $u$ and then $v$ to the DFA, $u$ and $v$ drive the DFA to the same state. From the point of view of the observer, $u$ and $v$ have the same effect (reaching the same state).

**Definition 2.3.** Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, we define the relation $\simeq_D$ on $\Sigma^*$ as follows: for any two strings $u, v \in \Sigma^*$,

$$u \simeq_D v \iff \delta^*(q_0, u) = \delta^*(q_0, v).$$
Example 2.1. We can figure out what the equivalence classes of $\simeq_D$ are for the following DFA:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

with 0 both start state and (unique) final state. For example

\[
abbabb \simeq_D aa \\
ababab \simeq_D \epsilon \\
bba \simeq_D a.
\]

There are three equivalences classes:

\[
[\epsilon], \enspace [a], \enspace [aa].
\]

Observe that $L(D) = [\epsilon]$. Also, the equivalence classes are in one-to-one correspondence with the states of $D$.

The relation $\simeq_D$ turns out to have some interesting properties. In particular, it is right-invariant, which means that for all $u, v, w \in \Sigma^*$, if $u \simeq v$, then $uw \simeq vw$.

**Proposition 2.6.** Given any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$, the relation $\simeq_D$ is an equivalence relation which is right-invariant and has finite index. Furthermore, if $Q$ has $n$ states, then the index of $\simeq_D$ is $n$, and every equivalence class of $\simeq_D$ is a regular language. Finally, $L(D)$ is the union of some of the equivalence classes of $\simeq_D$.

**Proof.** The fact that $\simeq_D$ is an equivalence relation is a trivial verification. To prove that $\simeq_D$ is right-invariant, we first prove by induction on the length of $v$ that for all $u, v \in \Sigma^*$, for all $p \in Q$,

\[
\delta^*(p, uv) = \delta^*(\delta^*(p, u), v).
\]

Then, if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, we have

\[
\delta^*(q_0, uv) = \delta^*(\delta^*(q_0, u), v) = \delta^*(\delta^*(q_0, v), w) = \delta^*(q_0, vw),
\]

which means that $uw \simeq_D vw$. Thus, $\simeq_D$ is right-invariant. We still have to prove that $\simeq_D$ has index $n$. Define the function $f : \Sigma^* \to Q$ such that

\[
f(u) = \delta^*(q_0, u).
\]

Note that if $u \simeq_D v$, which means that $\delta^*(q_0, u) = \delta^*(q_0, v)$, then $f(u) = f(v)$. Thus, the function $f : \Sigma^* \to Q$ has the same value on all the strings in some equivalence class $[u]$, so it induces a function $\widehat{f} : \Pi \to Q$ defined such that

\[
\widehat{f}([u]) = f(u),
\]
for every equivalence class \([u] \in \Pi\), where \(\Pi = \Sigma^*/\simeq\) is the partition associated with \(\simeq_D\).

However, the function \(\hat{f} : \Pi \to Q\) is injective (one-to-one), since \(\hat{f}([u]) = \hat{f}([v])\) is equivalent to \(f(u) = f(v)\) (since by definition of \(\hat{f}\) we have \(\hat{f}([u]) = f(u)\) and \(\hat{f}([v]) = f(v)\)), which by definition of \(f\) means that \(\delta^*(q_0, u) = \delta^*(q_0, v)\), which means precisely that \(u \simeq_D v\), that is, \([u] = [v]\).

Since \(Q\) has \(n\) states, \(\Pi\) has at most \(n\) blocks. Moreover, since every state is accessible, for every \(q \in Q\), there is some \(w \in \Sigma^*\) so that \(\delta^*(q_0, w) = q\), which shows that \(\hat{f}([w]) = f(w) = q\). Consequently, \(\hat{f}\) is also surjective. But then, being injective and surjective, \(\hat{f}\) is bijective and \(\Pi\) has exactly \(n\) blocks.

Every equivalence class of \(\Pi\) is a set of strings of the form
\[
\{w \in \Sigma^* \mid \delta^*(q_0, w) = p\},
\]
for some \(p \in Q\), which is accepted by the DFA
\[
D_p = (Q, \Sigma, \delta, q_0, \{p\})
\]
obtained from \(D\) by changing \(F\) to \(\{p\}\). Thus, every equivalence class is a regular language. Finally, since
\[
L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\} = \bigcup_{f \in F} \{w \in \Sigma^* \mid \delta^*(q_0, w) = f\} = \bigcup_{f \in F} L(D_f),
\]
we see that \(L(D)\) is the union of the equivalence classes corresponding to the final states in \(F\).

One should not be too optimistic and hope that every equivalence relation on strings is right-invariant.

**Example 2.2.** For example, if \(\Sigma = \{a\}\), the equivalence relation \(\simeq\) given by the partition
\[
\{\epsilon, a, a^2, a^3, a^4, a^5, a^6, a^7, a^8, \ldots \mid n \geq 0\} \cup \{a^{2^k}, a^{3^k}, a^{4^k}, a^{5^k}, a^{6^k}, a^{7^k}, a^{8^k}, \ldots \mid m \text{ is not a square}\}
\]
we have \(a \simeq a^4\), yet by concatenating on the right with \(a^5\), since \(aa^5 = a^6\) and \(a^4a^5 = a^9\) we get
\[
a^6 \not\simeq a^9,
\]
that is, \(a^6\) and \(a^9\) are not equivalent. It turns out that the problem is that neither equivalence class is a regular language.
It is worth noting that a right-invariant equivalence relation is not necessarily left-invariant, which means that if $u \simeq v$ then $wu \simeq vw$.

**Example 2.3.** For example, if $\simeq$ is given by the four equivalence classes

$$C_1 = \{bb\}^*, \ C_2 = \{bb\}^*a, \ C_3 = b\{bb\}^*, \ C_4 = \{bb\}^*a\{a, b\}^* \cup b\{bb\}^*a\{a, b\}^*,$$

then we can check that $\simeq$ is right-invariant by figuring out the inclusions $C_i a \subseteq C_j$ and $C_i b \subseteq C_j$, which are recorded in the following table:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>$C_2$</td>
<td>$C_3$</td>
</tr>
<tr>
<td>$C_2$</td>
<td>$C_4$</td>
<td>$C_4$</td>
</tr>
<tr>
<td>$C_3$</td>
<td>$C_4$</td>
<td>$C_1$</td>
</tr>
<tr>
<td>$C_4$</td>
<td>$C_4$</td>
<td>$C_4$</td>
</tr>
</tbody>
</table>

However, both $ab, ba \in C_4$, yet $bab \in C_4$ and $bba \in C_2$, so $\simeq$ is not left-invariant.

The remarkable fact due to Myhill and Nerode is that Proposition 2.6 has a converse. Indeed, given a right-invariant equivalence relation of finite index it is possible to reconstruct a DFA, and by a suitable choice of final state, every equivalence class is accepted by such a DFA. Let us show how this DFA is constructed using a simple example.

**Example 2.4.** Consider the equivalence relation $\simeq$ on $\{a, b\}^*$ given by the three equivalence classes

$$C_1 = \{\epsilon\}, \ C_2 = a\{a, b\}^*, \ C_3 = b\{a, b\}^*.$$

We leave it as an easy exercise to check that $\simeq$ is right-invariant. For example, if $u \simeq v$ and $u, v \in C_2$, then $u = ax$ and $v = ay$ for some $x, y \in \{a, b\}^*$, so for any $w \in \{a, b\}^*$ we have $uw = axw$ and $vw = ayw$, which means that we also have $uw, vw \in C_2$, thus $uw \simeq vw$.

For any subset $C \subseteq \{a, b\}^*$ and any string $w \in \{a, b\}^*$ define $Cw$ as the set of strings

$$Cw = \{uw \mid u \in C\}.$$ 

There are two reasons why a DFA can be recovered from the right-invariant equivalence relation $\simeq$:

1. For every equivalence class $C_i$ and every string $w$, there is a unique equivalence class $C_j$ such that $C_i w \subseteq C_j$.

Actually, it is enough to check the above property for strings $w$ of length 1 (i.e. symbols in the alphabet) because the property for arbitrary strings follows by induction.
2.5. RIGHT-INVARIANT EQUIVALENCE RELATIONS ON $\Sigma^*$

(2) For every $w \in \Sigma^*$ and every class $C_i$,

$$C_1w \subseteq C_i \iff w \in C_i,$$

where $C_1$ is the equivalence class of the empty string.

We can make a table recording these inclusions.

**Example 2.5.** Continuing Example 2.4, we get:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>$C_2$</td>
<td>$C_3$</td>
</tr>
<tr>
<td>$C_2$</td>
<td>$C_2$</td>
<td>$C_2$</td>
</tr>
<tr>
<td>$C_3$</td>
<td>$C_3$</td>
<td>$C_3$</td>
</tr>
</tbody>
</table>

For example, from $C_1 = \{\epsilon\}$ we have $C_1a = \{a\} \subseteq C_2$ and $C_1b = \{b\} \subseteq C_3$, for $C_2 = a\{a, b\}^*$, we have $C_2a = a\{a, b\}^*a \subseteq C_2$ and $C_2b = a\{a, b\}^*b \subseteq C_2$, and for $C_3 = b\{a, b\}^*$, we have $C_3a = b\{a, b\}^*a \subseteq C_3$ and $C_3b = b\{a, b\}^*b \subseteq C_3$.

The key point is that the above table is the transition table of a DFA with start state $C_1 = [\epsilon]$. Furthermore, if $C_i$ ($i = 1, 2, 3$) is chosen as a single final state, the corresponding DFA $D_i$ accepts $C_i$. This is the converse of Myhill-Nerode!

Observe that the inclusions $C_iw \subseteq C_j$ may be strict inclusions. For example, $C_1a = \{a\}$ is a proper subset of $C_2 = a\{a, b\}^*$.

Let us do another example.

**Example 2.6.** Consider the equivalence relation $\simeq$ given by the four equivalence classes

$$C_1 = \{\epsilon\}, \ C_2 = \{a\}, \ C_3 = \{b\}^+, \ C_4 = a\{a, b\}^+ \cup \{b\}^+a\{a, b\}^*.$$

We leave it as an easy exercise to check that $\simeq$ is right-invariant.

We obtain the following table of inclusions $C_i a \subseteq C_j$ and $C_i b \subseteq C_j$:

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>$C_2$</td>
<td>$C_3$</td>
</tr>
<tr>
<td>$C_2$</td>
<td>$C_4$</td>
<td>$C_4$</td>
</tr>
<tr>
<td>$C_3$</td>
<td>$C_4$</td>
<td>$C_3$</td>
</tr>
<tr>
<td>$C_4$</td>
<td>$C_4$</td>
<td>$C_4$</td>
</tr>
</tbody>
</table>

For example, from $C_3 = \{b\}^+$ we get $C_3a = \{b\}^+a \subseteq C_4$, and $C_3b = \{b\}^+b \subseteq C_3$.

The above table is the transition function of a DFA with four states and start state $C_1$. If $C_i$ ($i = 1, 2, 3, 4$) is chosen as a single final state, the corresponding DFA $D_i$ accepts $C_i$. 
Here is the general result.

**Proposition 2.7.** Given any equivalence relation \( \simeq \) on \( \Sigma^* \), if \( \simeq \) is right-invariant and has finite index \( n \), then every equivalence class (block) in the partition \( \Pi \) associated with \( \simeq \) is a regular language.

**Proof.** Let \( C_1, \ldots, C_n \) be the blocks of \( \Pi \), and assume that \( C_1 = [\epsilon] \) is the equivalence class of the empty string.

First, we claim that for every block \( C_i \) and every \( w \in \Sigma^* \), there is a unique block \( C_j \) such that \( C_iw \subseteq C_j \), where \( C_iw = \{uw \mid u \in C_i\} \).

For every \( u \in C_i \), the string \( uw \) belongs to one and only one of the blocks of \( \Pi \), say \( C_j \). For any other string \( v \in C_i \), since (by definition) \( u \simeq v \), by right invariance, we get \( uw \simeq vw \), but since \( uw \in C_j \) and \( C_j \) is an equivalence class, we also have \( vw \in C_j \). This proves the first claim.

We also claim that for every \( w \in \Sigma^* \), for every block \( C_i \),

\[
C_1w \subseteq C_i \iff w \in C_i.
\]

If \( C_1w \subseteq C_i \), since \( C_1 = [\epsilon] \), we have \( \epsilon w = w \in C_i \). Conversely, if \( w \in C_i \), for any \( v \in C_1 = [\epsilon] \), since \( \epsilon \simeq v \), by right invariance we have \( w \simeq vw \), and thus \( vw \in C_i \), which shows that \( C_1w \subseteq C_i \).

For every class \( C_k \), let

\[
D_k = (\{1, \ldots, n\}, \Sigma, \delta, 1, \{k\}),
\]

where \( \delta(i, a) = j \) iff \( C_i a \subseteq C_j \). We will prove the following equivalence:

\[
\delta^*(i, w) = j \iff C_iw \subseteq C_j.
\]

For this, we prove the following two implications by induction on \( |w| \):

(a) If \( \delta^*(i, w) = j \), then \( C_iw \subseteq C_j \), and

(b) If \( C_iw \subseteq C_j \), then \( \delta^*(i, w) = j \).

The base case \( (w = \epsilon) \) is trivial for both (a) and (b). We leave the proof of the induction step for (a) as an exercise and give the proof of the induction step for (b) because it is more subtle. Let \( w = ua \), with \( a \in \Sigma \) and \( u \in \Sigma^* \). If \( C_iua \subseteq C_j \), then by the first claim, we know that there is a unique block, \( C_k \), such that \( C_iu \subseteq C_k \). Furthermore, there is a unique block, \( C_h \), such that \( C_ha \subseteq C_h \), but \( C_iu \subseteq C_k \) implies \( C_iua \subseteq C_ha \) so we get \( C_iua \subseteq C_h \). However, by the uniqueness of the block, \( C_j \), such that \( C_iua \subseteq C_j \), we must have \( C_h = C_j \). By the induction hypothesis, as \( C_iu \subseteq C_k \), we have

\[
\delta^*(i, u) = k
\]
and, by definition of $\delta$, as $C_k a \subseteq C_j (= C_h)$, we have $\delta(k, a) = j$, so we deduce that
\[
\delta^*(i, ua) = \delta(\delta^*(i, u), a) = \delta(k, a) = j,
\]
as desired. Then, using the equivalence just proved and the second claim, we have
\[
L(D_k) = \{w \in \Sigma^* \mid \delta^*(1, w) \in \{k\}\} \\
= \{w \in \Sigma^* \mid \delta^*(1, w) = k\} \\
= \{w \in \Sigma^* \mid C_1 w \subseteq C_k\} \\
= \{w \in \Sigma^* \mid w \in C_k\} = C_k,
\]
proving that every block, $C_k$, is a regular language. 

In general it is false that $C_i a = C_j$ for some block $C_j$, and we can only claim that $C_i a \subseteq C_j$.

We can combine Proposition 2.6 and Proposition 2.7 to get the following characterization of a regular language due to Myhill and Nerode:

**Theorem 2.8.** (Myhill-Nerode) A language $L$ (over an alphabet $\Sigma$) is a regular language iff it is the union of some of the equivalence classes of an equivalence relation $\simeq$ on $\Sigma^*$, which is right-invariant and has finite index.

Theorem 2.8 can also be used to prove that certain languages are not regular. A general scheme (not the only one) goes as follows: If $L$ is not regular, then it must be infinite. Now, we argue by contradiction. If $L$ was regular, then by Myhill-Nerode, there would be some equivalence relation $\simeq$, which is right-invariant and of finite index, and such that $L$ is the union of some of the classes of $\simeq$. Because $\Sigma^*$ is infinite and $\simeq$ has only finitely many equivalence classes, there are strings $x, y \in \Sigma^*$ with $x \neq y$ so that
\[
x \simeq y.
\]
If we can find a third string, $z \in \Sigma^*$, such that
\[
xyz \in L \quad \text{and} \quad yz \notin L,
\]
then we reach a contradiction. Indeed, by right invariance, from $x \simeq y$, we get $xz \simeq yz$. But, $L$ is the union of equivalence classes of $\simeq$, so if $xz \in L$, then we should also have $yz \in L$, contradicting $yz \notin L$. Therefore, $L$ is not regular.

Then the scenario is this: to prove that $L$ is not regular, first we check that $L$ is infinite. If so, we try finding three strings $x, y, z$, where and $x$ and $y \neq x$ are prefixes of strings in $L$ such that
\[
x \simeq y,
\]
where \( \simeq \) is a right-invariant relation of finite index such that \( L \) is the union of equivalence of \( L \) (which must exist by Myhill–Nerode since we are assuming by contradiction that \( L \) is regular), and where \( z \) is chosen so that

\[
xz \in L \quad \text{and} \quad yz \notin L.
\]

**Example 2.7.** For example, we prove that \( L = \{a^n b^n \mid n \geq 1\} \) is not regular.

Assuming for the sake of contradiction that \( L \) is regular, there is some equivalence relation \( \simeq \) which is right-invariant and of finite index and such that \( L \) is the union of some of the classes of \( \simeq \). Since the sequence

\[
a, aa, aaa, \ldots, a^i, \ldots
\]

is infinite and \( \simeq \) has a finite number of classes, two of these strings must belong to the same class, which means that \( a^i \simeq a^j \) for some \( i \neq j \). But since \( \simeq \) is right invariant, by concatenating with \( b^i \) on the right, we see that \( a^i b^i \simeq a^j b^j \) for some \( i \neq j \). However \( a^i b^i \in L \), and since \( L \) is the union of classes of \( \simeq \), we also have \( a^j b^j \in L \) for \( i \neq j \), which is absurd, given the definition of \( L \). Thus, in fact, \( L \) is not regular.

Here is another illustration of the use of the Myhill-Nerode Theorem to prove that a language is not regular.

**Example 2.8.** We claim that the language,

\[
L' = \{a^n! \mid n \geq 1\},
\]

is not regular, where \( n! \) (\( n \) factorial) is given by \( 0! = 1 \) and \( (n + 1)! = (n + 1)n! \).

Assume \( L' \) is regular. Then, there is some equivalence relation \( \simeq \) which is right-invariant and of finite index and such that \( L' \) is the union of some of the classes of \( \simeq \). Since the sequence

\[
a, a^2, \ldots, a^n, \ldots
\]

is infinite, two of these strings must belong to the same class, which means that \( a^p \simeq a^q \) for some \( p, q \) with \( 1 \leq p < q \). As \( q! \geq q \) for all \( q \geq 0 \) and \( q > p \), we can concatenate on the right with \( a^{q-p} \) and we get

\[
a^p a^{q-p} \simeq a^q a^{q-p},
\]

that is,

\[
a^q \simeq a^{q+p-q-p}.
\]

Since \( p < q \) we have \( q! < q! + q - p \). If we can show that

\[
q! + q - p < (q + 1)!
\]

then
we will obtain a contradiction because then $a^{q!+q-p} \notin L'$, yet $a^{q!+q-p} \simeq a^{q!}$ and $a^{q!} \in L'$, contradicting Myhill-Nerode. Now, as $1 \leq p < q$, we have $q - p \leq q - 1$, so if we can prove that

$$q! + q - p \leq q! + q - 1 < (q + 1)!$$

we will be done. However, $q! + q - 1 < (q + 1)!$ is equivalent to

$$q - 1 < (q + 1)! - q!,$$

and since $(q + 1)! - q! = (q + 1)q! - q! = qq!$, we simply need to prove that

$$q - 1 < q \leq qq!,$$

which holds for $q \geq 1$.

There is another version of the Myhill-Nerode Theorem involving congruences which is also quite useful. An equivalence relation, $\simeq$, on $\Sigma^*$ is left and right-invariant iff for all $x, y, u, v \in \Sigma^*$,

$$\text{if } x \simeq y \text{ then } uxv \simeq wyv.$$

An equivalence relation, $\simeq$, on $\Sigma^*$ is a congruence iff for all $u_1, u_2, v_1, v_2 \in \Sigma^*$,

$$\text{if } u_1 \simeq v_1 \text{ and } u_2 \simeq v_2 \text{ then } u_1u_2 \simeq v_1v_2.$$

It is easy to prove that an equivalence relation is a congruence iff it is left and right-invariant.

For example, assume that $\simeq$ is a left and right-invariant equivalence relation, and assume that

$$u_1 \simeq v_1 \text{ and } u_2 \simeq v_2.$$

By right-invariance applied to $u_1 \simeq v_1$, we get

$$u_1u_2 \simeq v_1u_2$$

and by left-invariance applied to $u_2 \simeq v_2$ we get

$$v_1u_2 \simeq v_1v_2.$$

By transitivity, we conclude that

$$u_1u_2 \simeq v_1v_2.$$

which shows that $\simeq$ is a congruence.

Proving that a congruence is left and right-invariant is even easier.

There is a version of Proposition 2.6 that applies to congruences and for this we define the relation $\sim_D$ as follows: For any (trim) DFA, $D = (Q, \Sigma, \delta, q_0, F)$, for all $x, y \in \Sigma^*$,

$$x \sim_D y \iff (\forall q \in Q)(\delta^*(q, x) = \delta^*(q, y)).$$
Proposition 2.9. Given any (trim) DFA, \( D = (Q, \Sigma, \delta, q_0, F) \), the relation \( \sim_D \) is an equivalence relation which is left and right-invariant and has finite index. Furthermore, if \( Q \) has \( n \) states, then the index of \( \sim_D \) is at most \( n^n \) and every equivalence class of \( \sim_D \) is a regular language. Finally, \( L(D) \) is the union of some of the equivalence classes of \( \sim_D \).

Proof. We leave most of the proof of Proposition 2.9 as an exercise. The last two parts of the proposition are proved using the following facts:

(1) Since \( \sim_D \) is left and right-invariant and has finite index, in particular, \( \sim_D \) is left-invariant and has finite index, so by Proposition 2.7 every equivalence class of \( \sim_D \) is regular.

(2) Observe that 
\[
\sim_D \subseteq \simeq_D,
\]
since the condition \( \delta^*(q, x) = \delta^*(q, y) \) holds for every \( q \in Q \), so in particular for \( q = q_0 \). But then, every equivalence class of \( \simeq_D \) is the union of equivalence classes of \( \sim_D \) and since, by Proposition 2.6, \( L \) is the union of equivalence classes of \( \simeq_D \), we conclude that \( L \) is also the union of equivalence classes of \( \sim_D \).

This completes the proof. \( \square \)

Using Proposition 2.9 and Proposition 2.7, we obtain another version of the Myhill-Nerode Theorem.

Theorem 2.10. (Myhill-Nerode, Congruence Version) A language \( L \) (over an alphabet \( \Sigma \)) is a regular language iff it is the union of some of the equivalence classes of an equivalence relation \( \simeq \) on \( \Sigma^* \), which is a congruence and has finite index.

We now consider an equivalence relation associated with a language \( L \).

2.6 Finding minimal DFA’s

Given any language \( L \) (not necessarily regular), we can define an equivalence relation \( \rho_L \) on \( \Sigma^* \) which is right-invariant, but not necessarily of finite index. The equivalence relation \( \rho_L \) is such that \( L \) is the union of equivalence classes of \( \rho_L \). Furthermore, when \( L \) is regular, the relation \( \rho_L \) has finite index. In fact, this index is the size of a smallest DFA accepting \( L \). As a consequence, if \( L \) is regular, a simple modification of the proof of Proposition 2.7 applied to \( \simeq = \rho_L \) yields a minimal DFA \( D_{\rho_L} \) accepting \( L \).

Then, given any trim DFA \( D \) accepting \( L \), the equivalence relation \( \rho_L \) can be translated to an equivalence relation \( \equiv \) on states, in such a way that for all \( u, v \in \Sigma^* \),
\[
u \rho_L v \iff \varphi(u) \equiv \varphi(v),\]
where $\varphi : \Sigma^* \rightarrow Q$ is the function (run the DFA $D$ on $u$ from $q_0$) given by

$$\varphi(u) = \delta^*(q_0, u).$$

One can then construct a quotient DFA $D/ \equiv$ whose states are obtained by merging all states in a given equivalence class of states into a single state, and the resulting DFA $D/ \equiv$ is a minimal DFA. Even though $D/ \equiv$ appears to depend on $D$, it is in fact unique, and isomorphic to the abstract DFA $D_{\rho_L}$ induced by $\rho_L$.

The last step in obtaining the minimal DFA $D/ \equiv$ is to give a constructive method to compute the state equivalence relation $\equiv$. This can be done by constructing a sequence of approximations $\equiv_i$, where each $\equiv_{i+1}$ refines $\equiv_i$. It turns out that if $D$ has $n$ states, then there is some index $i_0 \leq n - 2$ such that

$$\equiv_j = \equiv_{i_0} \quad \text{for all } j \geq i_0 + 1,$$

and that

$$\equiv = \equiv_{i_0} .$$

Furthermore, $\equiv_{i+1}$ can be computed inductively from $\equiv_i$. In summary, we obtain a iterative algorithm for computing $\equiv$ that terminates in at most $n - 2$ steps.

**Definition 2.4.** Given any language $L$ (over $\Sigma$), we define the right-invariant equivalence $\rho_L$ associated with $L$ as the relation on $\Sigma^*$ defined as follows: for any two strings $u, v \in \Sigma^*$,

$$u \rho_L v \iff \forall w \in \Sigma^* (uw \in L \iff vw \in L).$$

It is clear that the relation $\rho_L$ is an equivalence relation, and it is right-invariant. To show right-invariance, argue as follows: if $u \rho_L v$, then for any $w \in \Sigma^*$, since $u \rho_L v$ means that

$$uz \in L \iff vz \in L$$

for all $z \in \Sigma^*$, in particular the above equivalence holds for all $z$ of the form $z = wy$ for any arbitrary $y \in \Sigma^*$, so we have

$$uwy \in L \iff vwz \in L$$

for all $y \in \Sigma^*$, which means that $uw \rho_L vw$.

It is also clear that $L$ is the union of the equivalence classes of strings in $L$. This is because if $u \in L$ and $u \rho_L v$, by letting $w = \epsilon$ in the definition of $\rho_L$, we get

$$u \in L \iff v \in L,$$

and since $u \in L$, we also have $v \in L$. This implies that if $u \in L$ then $[u]_{\rho_L} \subseteq L$ and so,

$$L = \bigcup_{u \in L} [u]_{\rho_L}.$$
Example 2.9. For example, consider the regular language
\[ L = \{a\} \cup \{b^m \mid m \geq 1\}. \]
We leave it as an exercise to show that the equivalence relation \( \rho_L \) consists of the four equivalence classes
\[ C_1 = \{\epsilon\}, \quad C_2 = \{a\}, \quad C_3 = \{b\}^+, \quad C_4 = a\{a, b\}^+ \cup \{b\}^+a\{a, b\}^* \]
en countered earlier in Example 2.6. Observe that
\[ L = C_2 \cup C_3. \]

When \( L \) is regular, we have the following remarkable result:

**Proposition 2.11.** Given any regular language \( L \), for any (trim) DFA \( D = (Q, \Sigma, \delta, q_0, F) \) such that \( L = L(D) \), \( \rho_L \) is a right-invariant equivalence relation, and we have \( \simeq_D \subseteq \rho_L \). Furthermore, if \( \rho_L \) has \( m \) classes and \( Q \) has \( n \) states, then \( m \leq n \).

**Proof.** By definition, \( u \simeq_D v \) iff \( \delta^*(q_0, u) = \delta^*(q_0, v) \). Since \( w \in L(D) \) iff \( \delta^*(q_0, w) \in F \), the fact that \( u \rho_L v \) can be expressed as
\[
\forall w \in \Sigma^*(uw \in L \iff vw \in L) \\
\text{iff} \\
\forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \iff \delta^*(q_0, vw) \in F) \\
\text{iff} \\
\forall w \in \Sigma^*(\delta^*(\delta^*(q_0, u), w) \in F \iff \delta^*(\delta^*(q_0, v), w) \in F),
\]
and if \( \delta^*(q_0, u) = \delta^*(q_0, v) \), this shows that \( u \rho_L v \). Since the number of classes of \( \simeq_D \) is \( n \) and \( \simeq_D \subseteq \rho_L \), the equivalence relation \( \rho_L \) has fewer classes than \( \simeq_D \), and \( m \leq n \).

Proposition 2.11 shows that when \( L \) is regular, the index \( m \) of \( \rho_L \) is finite, and it is a lower bound on the size of all DFA’s accepting \( L \). It remains to show that a DFA with \( m \) states accepting \( L \) exists.

However, going back to the proof of Proposition 2.7 starting with the right-invariant equivalence relation \( \rho_L \) of finite index \( m \), if \( L \) is the union of the classes \( C_{i_1}, \ldots, C_{i_k} \), the DFA
\[
D_{\rho_L} = (\{1, \ldots, m\}, \Sigma, \delta, 1, \{i_1, \ldots, i_k\}),
\]
where \( \delta(i, a) = j \) iff \( C_i a \subseteq C_j \), is such that \( L = L(D_{\rho_L}) \).

In summary, if \( L \) is regular, then the index of \( \rho_L \) is equal to the number of states of a minimal DFA for \( L \), and \( D_{\rho_L} \) is a minimal DFA accepting \( L \).
2.7. STATE EQUIVALENCE AND MINIMAL DFA’S

Example 2.10. For example, if

\[ L = \{a\} \cup \{b^m \mid m \geq 1\}. \]

then we saw in Example 2.9 that \( \rho_L \) consists of the four equivalence classes

\[ C_1 = \{\epsilon\}, \quad C_2 = \{a\}, \quad C_3 = \{b\}^+, \quad C_4 = a\{a,b\}^+ \cup \{b\}^+a\{a,b\}^+. \]

and we showed in Example 2.6 that the transition table of \( D_{\rho_L} \) is given by

\[
\begin{array}{c|cc}
  & a & b \\
\hline
C_1 & C_2 & C_3 \\
C_2 & C_4 & C_4 \\
C_3 & C_4 & C_3 \\
C_4 & C_4 & C_4 \\
\end{array}
\]

By picking the final states to be \( C_2 \) and \( C_3 \), we obtain the minimal DFA \( D_{\rho_L} \) accepting

\[ L = \{a\} \cup \{b^m \mid m \geq 1\}. \]

In the next section, we give an algorithm which allows us to find \( D_{\rho_L} \), given any DFA \( D \) accepting \( L \). This algorithms finds which states of \( D \) are equivalent.

2.7 State Equivalence and Minimal DFA’s

The proof of Proposition 2.11 suggests the following definition of an equivalence between states:

Definition 2.5. Given any DFA \( D = (Q, \Sigma, \delta, q_0, F) \), the relation \( \equiv \) on \( Q \), called state equivalence, is defined as follows: for all \( p, q \in Q \),

\[ p \equiv q \quad \text{iff} \quad \forall w \in \Sigma^* (\delta^*(p, w) \in F \quad \text{iff} \quad \delta^*(q, w) \in F). \]  

When \( p \equiv q \), we say that \( p \) and \( q \) are indistinguishable.

It is trivial to verify that \( \equiv \) is an equivalence relation, and that it satisfies the following property:

\[ \text{if } p \equiv q \text{ then } \delta(p, a) \equiv \delta(q, a), \quad \text{for all } a \in \Sigma. \]

To prove the above, since the condition defining \( \equiv \) must hold for all strings \( w \in \Sigma^* \), in particular it must hold for all strings of the form \( w = au \) with \( a \in \Sigma \) and \( u \in \Sigma^* \), so if \( p \equiv q \) then we have

\[
\begin{align*}
(\forall a \in \Sigma) (\forall u \in \Sigma^*) (\delta^*(p, au) \in F \quad \text{iff} \quad \delta^*(q, au) \in F) \\
\text{iff} \quad (\forall a \in \Sigma) (\forall u \in \Sigma^*) (\delta^*(\delta(p, a), u) \in F \quad \text{iff} \quad \delta^*(\delta(q, a), u) \in F) \\
\text{iff} \quad (\forall a \in \Sigma) (\forall u \in \Sigma^*) (\delta^*(\delta(p, a), u) \in F \quad \text{iff} \quad \delta^*(\delta(q, a), u) \in F) \\
\text{iff} \quad (\forall a \in \Sigma) (\delta(p, a) \equiv \delta(q, a)).
\end{align*}
\]
\[ \delta^*(p, \epsilon) \in F \text{ iff } \delta^*(q, \epsilon) \in F, \]

which, since \( \delta^*(p, \epsilon) = p \) and \( \delta^*(q, \epsilon) = q \), is equivalent to
\[ p \in F \text{ iff } q \in F. \]

Therefore, if two states \( p, q \) are equivalent, then either both \( p, q \in F \) or both \( p, q \in F \). This implies that a final state and a rejecting states are never equivalent.

**Example 2.11.** The reader should check that states \( A \) and \( C \) in the DFA below are equivalent and that no other distinct states are equivalent.

\[ \text{Figure 2.14: A non-minimal DFA for } \{a, b\}^* \{abb\} \]

It is illuminating to express state equivalence as the equality of two languages. Given the DFA \( D = (Q, \Sigma, \delta, q_0, F) \), let \( D_p = (Q, \Sigma, \delta, p, F) \) be the DFA obtained from \( D \) by redefining the start state to be \( p \). Then, it is clear that
\[ p \equiv q \text{ iff } L(D_p) = L(D_q). \]

This simple observation implies that there is an algorithm to test state equivalence. Indeed, we simply have to test whether the DFA’s \( D_p \) and \( D_q \) accept the same language and this can be done using the cross-product construction. Indeed, \( L(D_p) = L(D_q) \) iff \( L(D_p) - L(D_q) = \emptyset \) and \( L(D_q) - L(D_p) = \emptyset \). Now, if \( (D_p \times D_q)_{1-2} \) denotes the cross-product DFA with starting state \( (p, q) \) and with final states \( F \times (Q - F) \) and \( (D_p \times D_q)_{2-1} \) denotes the cross-product DFA also with starting state \( (p, q) \) and with final states \( (Q - F) \times F \), we know that
\[ L((D_p \times D_q)_{1-2}) = L((D_p \times D_q)_{2-1}) = L(D_q) - L(D_p), \]
so all we need to do if to test whether \( (D_p \times D_q)_{1-2} \) and \( (D_p \times D_q)_{2-1} \) accept the empty language. However, we know that this is the case iff the set of states reachable from \( (p, q) \)
in \((D_p \times D_q)_{1-2}\) contains no state in \(F \times (Q - F)\) and the set of states reachable from \((p, q)\)
in \((D_p \times D_q)_{2-1}\) contains no state in \((Q - F) \times F\).

Actually, the graphs of \((D_p \times D_q)_{1-2}\) and \((D_p \times D_q)_{2-1}\) are identical, so we only need to
cHECK that no state in \((F \times (Q - F)) \cup ((Q - F) \times F)\) is reachable from \((p, q)\) in that graph.

This algorithm to test state equivalence is not the most efficient but it is quite reasonable
(it runs in polynomial time).

If \(L = L(D)\), Theorem 2.12 below shows the relationship between \(\rho_L\) and \(\equiv\) and, more
generally, between the DFA, \(D_{\rho_L}\), and the DFA, \(D/\equiv\), obtained as the quotient of the DFA
\(D\) modulo the equivalence relation \(\equiv\) on \(Q\).

The minimal DFA \(D/\equiv\) is obtained by merging the states in each block \(C_i\) of the
partition \(\Pi\) associated with \(\equiv\), forming states corresponding to the blocks \(C_i\), and drawing
a transition on input \(a\) from a block \(C_i\) to a block \(C_j\) of \(\Pi\) iff there is a transition \(q = \delta(p, a)\)
from any state \(p \in C_i\) to any state \(q \in C_j\) on input \(a\).

The start state is the block containing \(q_0\), and the final states are the blocks consisting
of final states.

**Example 2.12.** For example, consider the DFA \(D_1\) accepting \(L = \{ab, ba\}^*\) shown in Figure
2.15.

\[
\begin{array}{c}
\text{0} \\
\text{1} \\
\text{3} \\
\text{4} \\
\text{2} \\
\text{5}
\end{array}
\]

\[
\begin{array}{c c c c}
\text{a} & \text{a} & \text{a} & \text{a} \\
\text{b} & \text{b} & \text{b} & \text{b} \\
\text{a} & \text{a} & \text{a} & \text{a} \\
\text{b} & \text{b} & \text{b} & \text{b} \\
\text{a} & \text{a} & \text{a} & \text{a} \\
\text{b} & \text{b} & \text{b} & \text{b}
\end{array}
\]

**Figure 2.15:** A nonminimal DFA \(D_1\) for \(L = \{ab, ba\}^*\)

This is not a minimal DFA. In fact,

\[0 \equiv 2 \quad \text{and} \quad 3 \equiv 5.\]

Here is the minimal DFA for \(L:\)

The minimal DFA \(D_2\) is obtained by merging the states in the equivalence class \(\{0, 2\}\)
into a single state, similarly merging the states in the equivalence class \(\{3, 5\}\) into a single
state, and drawing the transitions between equivalence classes. We obtain the DFA shown in Figure 2.16.

Formally, the quotient DFA $D/\equiv$ is defined such that

$$D/\equiv = (Q/\equiv, \Sigma, \delta/\equiv, [q_0]_\equiv, F/\equiv),$$

where

$$\delta/\equiv ([p]_\equiv, a) = [\delta(p, a)]_\equiv.$$

**Theorem 2.12.** For any (trim) DFA $D = (Q, \Sigma, \delta, q_0, F)$ accepting the regular language $L = L(D)$, the function $\varphi: \Sigma^* \rightarrow Q$ defined such that

$$\varphi(u) = \delta^*(q_0, u)$$

satisfies the property

$$u \rho_L v \iff \varphi(u) \equiv \varphi(v) \text{ for all } u, v \in \Sigma^*,$$

and induces a bijection $\hat{\varphi}: \Sigma^*/\rho_L \rightarrow Q/\equiv$, defined such that

$$\hat{\varphi}([u]_{\rho_L}) = [\delta^*(q_0, u)]_\equiv.$$

Furthermore, we have

$$[u]_{\rho_L} a \subseteq [v]_{\rho_L} \iff \delta(\varphi(u), a) \equiv \varphi(v).$$

Consequently, $\hat{\varphi}$ induces an isomorphism of DFA’s, $\hat{\varphi}: D_{\rho_L} \rightarrow D/\equiv$.

**Proof.** Since $\varphi(u) = \delta^*(q_0, u)$ and $\varphi(v) = \delta^*(q_0, v)$, the fact that $\varphi(u) \equiv \varphi(v)$ can be expressed as

$$\forall w \in \Sigma^*(\delta^*(q_0, u), w) \in F \iff \delta^*(\delta^*(q_0, v), w) \in F)$$

iff

$$\forall w \in \Sigma^*(\delta^*(q_0, uw) \in F \iff \delta^*(q_0, vw) \in F),$$
which is exactly \( u\rho_Lv \). Therefore,

\[ u\rho_Lv \iff \varphi(u) \equiv \varphi(v). \]

From the above, we see that the function \( \varphi : \Sigma^* \to Q \) maps each equivalence class \([u]\) modulo \( \rho_L \) to the equivalence class \([\varphi(u)]\) modulo \( \equiv \) and so, the function \( \widehat{\varphi} : \Sigma^*/\rho_L \to Q/ \equiv \) given by

\[ \widehat{\varphi}([u]_{\rho_L}) = [\delta^*(q_0, u)]_{\equiv} \]

is well-defined. Moreover, \( \widehat{\varphi} \) is injective, since \( \varphi([u]) = \varphi([v]) \iff \varphi(u) = \varphi(v) \iff (from \above) \ u\rho_Lv \iff [u] = [v] \). Since every state in \( Q \) is accessible, for every \( q \in Q \), there is some \( u \in \Sigma^* \) so that \( \varphi(u) = \delta^*(q_0, u) = q \), so \( \varphi([u]) = [q]_{\equiv} \) and \( \varphi \) is surjective. Therefore, we have a bijection \( \widehat{\varphi} : \Sigma^*/\rho_L \to Q/ \equiv \).

Since \( \varphi(u) = \delta^*(q_0, u) \), we have

\[ \delta(\varphi(u), a) = \delta(\delta^*(q_0, u), a) = \delta^*(q_0, ua) = \varphi(ua), \]

and thus, \( \delta(\varphi(u), a) \equiv \varphi(v) \) can be expressed as \( \varphi(ua) \equiv \varphi(v) \). By the previous part, this is equivalent to \( ua\rho_Lv \), and we claim that this is equivalent to

\[ [u]_{\rho_L}a \subseteq [v]_{\rho_L}. \]

First, if \( [u]_{\rho_L}a \subseteq [v]_{\rho_L} \), then \( ua \in [v]_{\rho_L} \), that is, \( ua\rho_Lv \). Conversely, if \( ua\rho_Lv \), then for every \( u' \in [u]_{\rho_L} \), we have \( u'a\rho_Lua \), so by right-invariance we get \( u'a\rho_Lua \), and since \( ua\rho_Lv \), we get \( u'a\rho_Lv \), that is, \( u'a \in [v]_{\rho_L} \). Since \( u' \in [u]_{\rho_L} \) is arbitrary, we conclude that \( [u]_{\rho_L}a \subseteq [v]_{\rho_L} \).

Therefore, we proved that

\[ \delta(\varphi(u), a) \equiv \varphi(v) \iff [u]_{\rho_L}a \subseteq [v]_{\rho_L}. \]

The above shows that the transitions of \( D_{\rho_L} \) correspond to the transitions of \( D/ \equiv \). \( \square \)

Theorem 2.12 shows that the DFA \( D_{\rho_L} \) is isomorphic to the DFA \( D/ \equiv \) obtained as the quotient of the DFA \( D \) modulo the equivalence relation \( \equiv \) on \( Q \). Since \( D_{\rho_L} \) is a minimal DFA accepting \( L \), so is \( D/ \equiv \).

**Example 2.13.** Consider the following DFA \( D \),

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

with start state 1 and final states 2 and 3. It is easy to see that

\[ L(D) = \{a\} \cup \{b^m \mid m \geq 1\} . \]
It is not hard to check that states 4 and 5 are equivalent, and no other pairs of distinct states are equivalent. The quotient DFA $D/ \equiv$ is obtained by merging states 4 and 5, and we obtain the following minimal DFA:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

with start state 1 and final states 2 and 3. This DFA is isomorphic to the DFA $D_{\rho_L}$ of Example 2.10.

There are other characterizations of the regular languages. Among those, the characterization in terms of right derivatives is of particular interest because it yields an alternative construction of minimal DFA’s.

**Definition 2.6.** Given any language, $L \subseteq \Sigma^*$, for any string, $u \in \Sigma^*$, the right derivative of $L$ by $u$, denoted $L/u$, is the language

$$L/u = \{ w \in \Sigma^* \mid uw \in L \}.$$ 

**Theorem 2.13.** If $L \subseteq \Sigma^*$ is any language, then $L$ is regular iff it has finitely many right derivatives. Furthermore, if $L$ is regular, then all its right derivatives are regular and their number is equal to the number of states of the minimal DFA’s for $L$.

**Proof.** It is easy to check that

$$L/u = L/v \quad \text{iff} \quad u \rho_L v.$$ 

The above shows that $\rho_L$ has a finite number of classes, say $m$, iff there is a finite number of right derivatives, say $n$, and if so, $m = n$. If $L$ is regular, then we know that the number of equivalence classes of $\rho_L$ is the number of states of the minimal DFA’s for $L$, so the number of right derivatives of $L$ is equal to the size of the minimal DFA’s for $L$.

Conversely, if the number of derivatives is finite, say $m$, then $\rho_L$ has $m$ classes and by Myhill-Nerode, $L$ is regular. It remains to show that if $L$ is regular then every right derivative is regular.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA accepting $L$. If $p = \delta^*(q_0, u)$, then let

$$D_p = (Q, \Sigma, \delta, p, F),$$

that is, $D$ with $p$ as start state. It is clear that

$$L/u = L(D_p),$$

so $L/u$ is regular for every $u \in \Sigma^*$. Also observe that if $|Q| = n$, then there are at most $n$ DFA’s $D_p$, so there is at most $n$ right derivatives, which is another proof of the fact that a regular language has a finite number of right derivatives. $\square$
If $L$ is regular then the construction of a minimal DFA for $L$ can be recast in terms of right derivatives. Let $L/u_1, L/u_2, \ldots, L/u_m$ be the set of all the right derivatives of $L$. Of course, we may assume that $u_1 = \epsilon$. We form a DFA whose states are the right derivatives, $L/u_i$. For every state, $L/u_i$, for every $a \in \Sigma$, there is a transition on input $a$ from $L/u_i$ to $L/u_j = L/(u_ia)$. The start state is $L = L/u_1$ and the final states are the right derivatives, $L/u_i$, for which $\epsilon \in L/u_i$.

We leave it as an exercise to check that the above DFA accepts $L$. One way to do this is to recall that $L/u_i = L/v_i$ if $u \rho L v$ and to observe that the above construction mimics the construction of $D_{\rho L}$ as in the Myhill-Nerode proposition (Proposition 2.7). This DFA is minimal since the number of right derivatives is equal to the size of the minimal DFA’s for $L$.

We now return to state equivalence. Note that if $F = \emptyset$, then $\equiv$ has a single block ($Q$), and if $F = Q$, then $\equiv$ has a single block ($F$). In the first case, the minimal DFA is the one state DFA rejecting all strings. In the second case, the minimal DFA is the one state DFA accepting all strings. When $F \neq \emptyset$ and $F \neq Q$, there are at least two states in $Q$, and $\equiv$ also has at least two blocks, as we shall see shortly.

It remains to compute $\equiv$ explicitly. This is done using a sequence of approximations. In view of the previous discussion, we are assuming that $F \neq \emptyset$ and $F \neq Q$, which means that $n \geq 2$, where $n$ is the number of states in $Q$.

**Definition 2.7.** Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, for every $i \geq 0$, the relation $\equiv_i$ on $Q$, called $i$-state equivalence, is defined as follows: for all $p, q \in Q$,

$$p \equiv_i q \text{iff } \forall w \in \Sigma^*, |w| \leq i (\delta^*(p, w) \in F \text{ iff } \delta^*(q, w) \in F).$$

When $p \equiv_i q$, we say that $p$ and $q$ are $i$-indistinguishable.

Since state equivalence $\equiv$ is defined such that

$$p \equiv q \text{iff } \forall w \in \Sigma^*(\delta^*(p, w) \in F \text{ iff } \delta^*(q, w) \in F),$$

we note that testing the condition

$$\delta^*(p, w) \in F \text{ iff } \delta^*(q, w) \in F$$

for all strings in $\Sigma^*$ is equivalent to testing the above condition for all strings of length at most $i$ for all $i \geq 0$, i.e.

$$p \equiv q \text{iff } \forall i \geq 0 \forall w \in \Sigma^*, |w| \leq i (\delta^*(p, w) \in F \text{ iff } \delta^*(q, w) \in F).$$

Since $\equiv_i$ is defined such that

$$p \equiv_i q \text{iff } \forall w \in \Sigma^*, |w| \leq i (\delta^*(p, w) \in F \text{ iff } \delta^*(q, w) \in F),$$
we conclude that
\[ p \equiv q \iff \forall i \geq 0 (p \equiv_i q). \]
This identity can also be expressed as
\[ \equiv = \bigcap_{i \geq 0} \equiv_i. \]

If we assume that \( F \neq \emptyset \) and \( F \neq Q \), observe that \( \equiv_0 \) has exactly two equivalence classes \( F \) and \( Q - F \), since \( \epsilon \) is the only string of length 0, and since the condition
\[ \delta^*(p, \epsilon) \in F \iff \delta^*(q, \epsilon) \in F \]
is equivalent to the condition
\[ p \in F \iff q \in F. \]
It is also obvious from the definition of \( \equiv_i \) that
\[ \equiv \subseteq \cdots \subseteq \equiv_{i+1} \subseteq \equiv_i \subseteq \cdots \subseteq \equiv_1 \subseteq \equiv_0. \]
If this sequence was strictly decreasing for all \( i \geq 0 \), the partition associated with \( \equiv_{i+1} \) would contain at least one more block than the partition associated with \( \equiv_i \) and since we start with a partition with two blocks, the partition associated with \( \equiv_i \) would have at least \( i + 2 \) blocks. But then, for \( i = n - 1 \), the partition associated with \( \equiv_{n-1} \) would have at least \( n + 1 \) blocks, which is absurd since \( Q \) has only \( n \) states. Therefore, there is a smallest integer, \( i_0 \leq n - 2 \), such that
\[ \equiv_{i_0+1} = \equiv_{i_0}. \]
Thus, it remains to compute \( \equiv_{i+1} \) from \( \equiv_i \), which can be done using the following proposition: The proposition also shows that
\[ \equiv = \equiv_{i_0}. \]

**Proposition 2.14.** For any (trim) DFA \( D = (Q, \Sigma, \delta, q_0, F) \), for all \( p, q \in Q \), \( p \equiv_{i+1} q \) iff \( p \equiv_i q \) and \( \delta(p, a) \equiv_i \delta(q, a) \), for every \( a \in \Sigma \). Furthermore, if \( F \neq \emptyset \) and \( F \neq Q \), there is a smallest integer \( i_0 \leq n - 2 \), such that
\[ \equiv_{i_0+1} = \equiv_{i_0} = \equiv. \]

**Proof.** By the definition of the relation \( \equiv_i \),
\[ p \equiv_{i+1} q \iff \forall w \in \Sigma^*, |w| \leq i + 1 (\delta^*(p, w) \in F \iff \delta^*(q, w) \in F). \]
The trick is to observe that the condition
\[ \delta^*(p, w) \in F \iff \delta^*(q, w) \in F \]
holds for all strings of length at most $i + 1$ iff it holds for all strings of length at most $i$ and for all strings of length between 1 and $i + 1$. This is expressed as

$$p \equiv_{i+1} q \iff \forall w \in \Sigma^*, |w| \leq i \ (\delta^*(p,w) \in F \iff \delta^*(q,w) \in F)$$

and

$$\forall w \in \Sigma^*, 1 \leq |w| \leq i + 1 \ (\delta^*(p,w) \in F \iff \delta^*(q,w) \in F).$$

Obviously, the first condition in the conjunction is $p \equiv_i q$, and since every string $w$ such that $1 \leq |w| \leq i + 1$ can be written as $au$ where $a \in \Sigma$ and $0 \leq |u| \leq i$, the second condition in the conjunction can be written as

$$\forall a \in \Sigma \forall u \in \Sigma^*, |u| \leq i \ (\delta^*(p,au) \in F \iff \delta^*(q,au) \in F).$$

However, $\delta^*(p,au) = \delta^*(\delta(p,a),u)$ and $\delta^*(q,au) = \delta^*(\delta(q,a),u)$, so that the above condition is really

$$\forall a \in \Sigma \ (\delta(p,a) \equiv_i \delta(q,a)).$$

Thus, we showed that

$$p \equiv_{i+1} q \iff p \equiv_i q \text{ and } \forall a \in \Sigma \ (\delta(p,a) \equiv_i \delta(q,a)).$$

Thus, if $\equiv_{i+1} = \equiv_i$ for some $i \geq 0$, using induction, we also have $\equiv_{i+j} = \equiv_i$ for all $j \geq 1$. Since

$$\equiv = \bigcap_{i \geq 0} \equiv_i, \quad \equiv_{i+1} \subseteq \equiv_i,$$

and since we know that there is a smallest index say $i_0$, such that $\equiv_j = \equiv_{i_0}$, for all $j \geq i_0 + 1$, we have $\equiv = \equiv_{i_0}$.

Using Proposition 2.14, we can compute $\equiv$ inductively, starting from $\equiv_0 = (F, Q - F)$, and computing $\equiv_{i+1}$ from $\equiv_i$, until the sequence of partitions associated with the $\equiv_i$ stabilizes.

Note that if $F = Q$ or $F = \emptyset$, then $\equiv = \equiv_0$, and the inductive characterization of Proposition 2.14 holds trivially.

There are a number of algorithms for computing $\equiv$, or to determine whether $p \equiv q$ for some given $p, q \in Q$.

A simple method to compute $\equiv$ is described in Hopcroft and Ullman. The basic idea is to propagate inequivalence, rather than equivalence.

The method consists in forming a triangular array corresponding to all unordered pairs $(p, q)$, with $p \neq q$ (the rows and the columns of this triangular array are indexed by the states in $Q$, where the entries are below the descending diagonal). Initially, the entry $(p, q)$ is marked iff $p$ and $q$ are not 0-equivalent, which means that $p$ and $q$ are not both in $F$ or not both in $Q - F$. 
Then, we process every unmarked entry on every row as follows: for any unmarked pair \((p, q)\), we consider pairs \((\delta(p, a), \delta(q, a))\), for all \(a \in \Sigma\). If any pair \((\delta(p, a), \delta(q, a))\) is already marked, this means that \(\delta(p, a)\) and \(\delta(q, a)\) are inequivalent, and thus \(p\) and \(q\) are inequivalent, and we mark the pair \((p, q)\). We continue in this fashion, until at the end of a round during which all the rows are processed, nothing has changed. When the algorithm stops, all marked pairs are inequivalent, and all unmarked pairs correspond to equivalent states.

Let us illustrate the above method.

**Example 2.14.** Consider the following DFA accepting \(\{a, b\}^*\{abb\}^*\):

\[
\begin{array}{|c|c|c|}
\hline
& a & b \\
\hline
A & B & C \\
B & B & D \\
C & B & C \\
D & B & E \\
E & B & C \\
\hline
\end{array}
\]

The start state is \(A\), and the set of final states is \(F = \{E\}\). (This is the DFA displayed in Figure 2.7.)

The initial (half) array is as follows, using \(\times\) to indicate that the corresponding pair (say, \((E, A)\)) consists of inequivalent states, and \(\square\) to indicate that nothing is known yet.

\[
\begin{array}{c}
B \quad \square \\
C \quad \square \quad \square \\
D \quad \square \quad \square \quad \square \\
E \times \times \times \times \\
A \quad B \quad C \quad D
\end{array}
\]

After the first round, we have

\[
\begin{array}{c}
B \quad \square \\
C \quad \square \quad \square \\
D \times \times \times \\
E \times \times \times \times \\
A \quad B \quad C \quad D
\end{array}
\]

After the second round, we have

\[
\begin{array}{c}
B \times \\
C \quad \square \times \\
D \times \times \times \\
E \times \times \times \times \\
A \quad B \quad C \quad D
\end{array}
\]
Finally, nothing changes during the third round, and thus, only $A$ and $C$ are equivalent, and we get the four equivalence classes

$$\{(A, C), \{B\}, \{D\}, \{E\}\}.$$ We obtain the minimal DFA showed in Figure 2.17.

![Figure 2.17: A minimal DFA accepting $\{a, b\}^*\{abb\}$](image)

There are ways of improving the efficiency of this algorithm, see Hopcroft and Ullman for such improvements. Fast algorithms for testing whether $p \equiv q$ for some given $p, q \in Q$ also exist. One of these algorithms is based on “forward closures,” following an idea of Knuth. Such an algorithm is related to a fast unification algorithm; see Section 2.9.

### 2.8 The Pumping Lemma

Another useful tool for proving that languages are not regular is the so-called pumping lemma.

**Proposition 2.15.** (Pumping lemma) Given any DFA $D = (Q, \Sigma, \delta, q_0, F)$, there is some $m \geq 1$ such that for every $w \in \Sigma^*$, if $w \in L(D)$ and $|w| \geq m$, then there exists a decomposition of $w$ as $w = uxv$, where

1. $x \neq \epsilon$,
2. $ux^iv \in L(D)$, for all $i \geq 0$, and
3. $|ux| \leq m$.

Moreover, $m$ can be chosen to be the number of states of the DFA $D$. 
Proof. Let \( m \) be the number of states in \( Q \), and let \( w = w_1 \ldots w_n \). Since \( Q \) contains the start state \( q_0 \), \( m \geq 1 \). Since \( |w| \geq m \), we have \( n \geq m \). Since \( w \in L(D) \), let \((q_0, q_1, \ldots, q_n)\), be the sequence of states in the accepting computation of \( w \) (where \( q_n \in F \)). Consider the subsequence 
\((q_0, q_1, \ldots, q_m)\).

This sequence contains \( m + 1 \) states, but there are only \( m \) states in \( Q \), and thus, we have \( q_i = q_j \) for some \( i, j \) such that \( 0 \leq i < j \leq m \). Then, letting \( u = w_1 \ldots w_i \), \( x = w_{i+1} \ldots w_j \), and \( v = w_{j+1} \ldots w_n \), it is clear that the conditions of the proposition hold.

An important consequence of the pumping lemma is that if a DFA \( D \) has \( m \) states and if there is some string \( w \in L(D) \) such that \( |w| \geq m \), then \( L(D) \) is infinite.

Indeed, by the pumping lemma, \( w \in L(D) \) can be written as \( w = uxv \) with \( x \neq \epsilon \), and
\[
ux^i v \in L(D) \quad \text{for all} \ i \geq 0.
\]

Since \( x \neq \epsilon \), we have \( |x| > 0 \), so for all \( i, j \geq 0 \) with \( i < j \) we have
\[
|ux^i v| < |ux^i v| + (j - i)|x| = |ux^j v|,
\]
which implies that \( ux^i v \neq ux^j v \) for all \( i < j \), and the set of strings
\[
\{ux^i v \mid i \geq 0\} \subseteq L(D)
\]
is an infinite subset of \( L(D) \), which is itself infinite.

As a consequence, if \( L(D) \) is finite, there are no strings \( w \) in \( L(D) \) such that \( |w| \geq m \). In this case, since the premise of the pumping lemma is false, the pumping lemma holds vacuously; that is, if \( L(D) \) is finite, the pumping lemma yields no information.

Another corollary of the pumping lemma is that there is a test to decide whether a DFA \( D \) accepts an infinite language \( L(D) \).

**Proposition 2.16.** Let \( D \) be a DFA with \( m \) states, The language \( L(D) \) accepted by \( D \) is infinite iff there is some string \( w \in L(D) \) such that \( m \leq |w| < 2m \).

If \( L(D) \) is infinite, there are strings of length \( \geq m \) in \( L(D) \), but a priori there is no guarantee that there are “short” strings \( w \) in \( L(D) \), that is, strings whose length is uniformly bounded by some function of \( m \) independent of \( D \). The pumping lemma ensures that there are such strings, and the function is \( m \mapsto 2m \).

Typically, the pumping lemma is used to prove that a language is not regular. The method is to proceed by contradiction, i.e., to assume (contrary to what we wish to prove) that a language \( L \) is indeed regular, and derive a contradiction of the pumping lemma. Thus,
it would be helpful to see what the negation of the pumping lemma is, and for this, we first state the pumping lemma as a logical formula. We will use the following abbreviations:

- \( nat = \{0, 1, 2, \ldots\} \),
- \( pos = \{1, 2, \ldots\} \),
- \( A \equiv w = uxv \),
- \( B \equiv x \neq \epsilon \),
- \( C \equiv |ux| \leq m \),
- \( P \equiv \forall i: nat \ (ux^i v \in L(D)) \).

The pumping lemma can be stated as

\[ \forall D: \text{DFA } \exists m: pos \forall w: \Sigma^* \left( (w \in L(D) \land |w| \geq m) \supset (\exists u, x, v: \Sigma^* A \land B \land C \land P) \right). \]

Recalling that

\[ \neg (A \land B \land C \land P) \equiv \neg (A \land B \land C) \lor \neg P \equiv (A \land B \land C) \supset \neg P \]

and

\[ \neg (R \supset S) \equiv R \land \neg S, \]

the negation of the pumping lemma can be stated as

\[ \exists D: \text{DFA } \forall m: pos \exists w: \Sigma^* \left( (w \in L(D) \land |w| \geq m) \land (\forall u, x, v: \Sigma^* (A \land B \land C) \supset \neg P) \right). \]

Since

\[ \neg P \equiv \exists i: nat \ (ux^i v \notin L(D)), \]

in order to show that the pumping lemma is contradicted, one needs to show that for some DFA \( D \), for every \( m \geq 1 \), there is some string \( w \in L(D) \) of length at least \( m \), such that for every possible decomposition \( w = uxv \) satisfying the constraints \( x \neq \epsilon \) and \( |ux| \leq m \), there is some \( i \geq 0 \) such that \( ux^i v \notin L(D) \).

When proceeding by contradiction, we have a language \( L \) that we are (wrongly) assuming to be regular, and we can use any DFA \( D \) accepting \( L \). The creative part of the argument is to pick the right \( w \in L \) (not making any assumption on \( m \leq |w| \)).

As an illustration, let us use the pumping lemma to prove that \( L_1 = \{a^n b^n \mid n \geq 1\} \) is not regular. The usefulness of the condition \( |ux| \leq m \) lies in the fact that it reduces the number of legal decomposition \( uxv \) of \( w \). We proceed by contradiction. Thus, let us assume that \( L_1 = \{a^n b^n \mid n \geq 1\} \) is regular. If so, it is accepted by some DFA \( D \). Now, we wish to contradict the pumping lemma. For every \( m \geq 1 \), let \( w = a^m b^m \). Clearly, \( w = a^m b^m \in L_1 \) and \( |w| \geq m \). Then, every legal decomposition \( u, x, v \) of \( w \) is such that

\[ w = a \ldots a \ a \ldots a b \ldots b \]
where \( x \neq \epsilon \) and \( x \) ends within the \( a \)'s, since \( |ux| \leq m \). Since \( x \neq \epsilon \), the string \( uxxv \) is of the form \( a^n b^m \) where \( n > m \), and thus \( uxxv \notin L_1 \), contradicting the pumping lemma.

Let us consider two more examples. Let \( L_2 = \{ a^m b^n \mid 1 \leq m < n \} \). We claim that \( L_2 \) is not regular. Our first proof uses the pumping lemma. For any \( m \geq 1 \), pick \( w = a^m b^{m+1} \). We have \( w \in L_2 \) and \( |w| \geq m \) so we need to contradict the pumping lemma. Every legal decomposition \( u, x, v \) of \( w \) is such that

\[
    w = a \ldots a a \ldots a a \ldots ab \ldots b
\]

where \( x \neq \epsilon \) and \( x \) ends within the \( a \)'s, since \( |ux| \leq m \). Since \( x \neq \epsilon \), the string \( uxxv = uxxv \) contains at least \( m+1 \) \( a \)'s and still \( m+1 \) \( b \)'s, so \( uxxv \notin L_2 \), contradicting the pumping lemma.

Our second proof uses Myhill-Nerode. Let \( \equiv \) be a right-invariant equivalence relation of finite index such that \( L_2 \) is the union of classes of \( \equiv \). If we consider the infinite sequence

\[
a, a^2, \ldots, a^n, \ldots
\]

since \( \equiv \) has a finite number of classes there are two strings \( a^m \) and \( a^n \) with \( m < n \) such that

\[
a^m \equiv a^n.
\]

By right-invariance by concatenating on the right with \( b^n \) we obtain

\[
a^m b^n \equiv a^n b^n,
\]

and since \( m < n \) we have \( a^m b^n \in L_2 \) but \( a^n b^n \notin L_2 \), a contradiction.

Let us now consider the language \( L_3 = \{ a^m b^n \mid m \neq n \} \). This time let us begin by using Myhill-Nerode to prove that \( L_3 \) is not regular. The proof is the same as before, we obtain

\[
a^m b^n \equiv a^n b^n,
\]

and the contradiction is that \( a^m b^n \in L_3 \) and \( a^n b^n \notin L_3 \).

Let use now try to use the pumping lemma to prove that \( L_3 \) is not regular. For any \( m \geq 1 \) pick \( w = a^m b^{m+1} \in L_3 \). As in the previous case, every legal decomposition \( u, x, v \) of \( w \) is such that

\[
w = a \ldots a a \ldots a a \ldots ab \ldots b
\]

where \( x \neq \epsilon \) and \( x \) ends within the \( a \)'s, since \( |ux| \leq m \). However this time we have a problem, namely that we know that \( x \) is a nonempty string of \( a \)'s but we don’t know how many, so we can’t guarantee that pumping up \( x \) will yield exactly the string \( a^{m+1} b^{m+1} \). We made the wrong choice for \( w \). There is a choice that will work but it is a bit tricky.
2.9. A FAST ALGORITHM FOR CHECKING STATE EQUIVALENCE

Fortunately, there is another simpler approach. Recall that the regular languages are closed under the boolean operations (union, intersection and complementation). Thus, $L_3$ is not regular iff its complement $\overline{L_3}$ is not regular. Observe that $\overline{L_3}$ contains $\{a^nb^n \mid n \geq 1\}$, which we showed to be nonregular. But there is another problem, which is that $L_3$ contains other strings besides strings of the form $a^nb^n$, for example strings of the form $b^ma^n$ with $m, n > 0$.

Again, we can take care of this difficulty using the closure operations of the regular languages. If we can find a regular language $R$ such that $L_3 \cap R$ is not regular, then $L_3$ itself is not regular, since otherwise as $\overline{L_3}$ and $R$ are regular then $\overline{L_3} \cap R$ is also regular. In our case, we can use $R = \{a\}^+\{b\}^+$ to obtain

$$L_3 \cap \{a\}^+\{b\}^+ = \{a^nb^n \mid n \geq 1\}.$$  

Since $\{a^nb^n \mid n \geq 1\}$ is not regular, we reached our final contradiction. Observe how we use the language $R$ to “clean up” $\overline{L_3}$ by intersecting it with $R$.

To complete a direct proof using the pumping lemma, the reader should try $w = a^mb^{(m+1)!}$.

The use of the closure operations of the regular languages is often a quick way of showing that a language $L$ is not regular by reducing the problem of proving that $L$ is not regular to the problem of proving that some well-known language is not regular.

2.9 A Fast Algorithm for Checking State Equivalence

Using a “Forward-Closure”

Given two states $p, q \in Q$, if $p \equiv q$, then we know that $\delta(p, a) \equiv \delta(q, a)$, for all $a \in \Sigma$. This suggests a method for testing whether two distinct states $p, q$ are equivalent. Starting with the relation $R = \{(p, q)\}$, construct the smallest equivalence relation $R^*$ containing $R$ with the property that whenever $(r, s) \in R^*$, then $(\delta(r, a), \delta(s, a)) \in R^*$, for all $a \in \Sigma$. If we ever encounter a pair $(r, s)$ such that $r \in F$ and $s \in \overline{F}$, or $r \in \overline{F}$ and $s \in F$, then $r$ and $s$ are inequivalent, and so are $p$ and $q$. Otherwise, it can be shown that $p$ and $q$ are indeed equivalent. Thus, testing for the equivalence of two states reduces to finding an efficient method for computing the “forward closure” of a relation defined on the set of states of a DFA.

Such a method was worked out by John Hopcroft and Richard Karp and published in a 1971 Cornell technical report. This method is based on an idea of Donald Knuth for solving Exercise 11, in Section 2.3.5 of *The Art of Computer Programming*, Vol. 1, second edition, 1973. A sketch of the solution for this exercise is given on page 594. As far as I know, Hopcroft and Karp’s method was never published in a journal, but a simple recursive algorithm does appear on page 144 of Aho, Hopcroft and Ullman’s *The Design and Analysis of Computer Algorithms*, first edition, 1974. Essentially the same idea was used by Paterson and Wegman to design a fast unification algorithm (in 1978). We make a few definitions.
A relation \( S \subseteq Q \times Q \) is a **forward closure** if it is an equivalence relation and whenever \((r, s) \in S\), then \((\delta(r, a), \delta(s, a)) \in S\), for all \( a \in \Sigma \). The **forward closure** of a relation \( R \subseteq Q \times Q \) is the smallest equivalence relation \( R^\dagger \) containing \( R \) which is forward closed.

We say that a forward closure \( S \) is **good** if whenever \((r, s) \in S\), then \( \text{good}(r, s) \) holds iff either both \( r, s \in F \), or both \( r, s \notin F \). Obviously, \( \text{bad}(r, s) \) iff \( \neg \text{good}(r, s) \).

Given any relation \( R \subseteq Q \times Q \), recall that the smallest equivalence relation \( R \approx \) containing \( R \) is the relation \( (R \cup R^{-1})^* \) (where \( R^{-1} = \{ (q, p) \mid (p, q) \in R \} \)), and \((R \cup R^{-1})^*\) is the reflexive and transitive closure of \((R \cup R^{-1})\). The forward closure of \( R \) can be computed inductively by defining the sequence of relations \( R_i \subseteq Q \times Q \) as follows:

\[
R_0 = R \approx \\
R_{i+1} = (R_i \cup \{ (\delta(r, a), \delta(s, a)) \mid (r, s) \in R_i, \ a \in \Sigma \}) \approx.
\]

It is not hard to prove that \( R_{i_0+1} = R_{i_0} \) for some least \( i_0 \), and that \( R^\dagger = R_{i_0} \) is the smallest forward closure containing \( R \). The following two facts can also been established.

(a) if \( R^\dagger \) is good, then

\[
R^\dagger \subseteq \equiv.
\]  \hspace{1cm} \text{(2.1)}

(b) if \( p \equiv q \), then

\[
R^\dagger \subseteq \equiv,
\]

that is, equation (2.1) holds. This implies that \( R^\dagger \) is good.

As a consequence, we obtain the correctness of our procedure: \( p \equiv q \) iff the forward closure \( R^\dagger \) of the relation \( R = \{(p, q)\} \) is good.

In practice, we maintain a partition \( \Pi \) representing the equivalence relation that we are closing under forward closure. We add each new pair \((\delta(r, a), \delta(s, a))\) one at a time, and immediately form the smallest equivalence relation containing the new relation. If \( \delta(r, a) \) and \( \delta(s, a) \) already belong to the same block of \( \Pi \), we consider another pair, else we merge the blocks corresponding to \( \delta(r, a) \) and \( \delta(s, a) \), and then consider another pair.

The algorithm is recursive, but it can easily be implemented using a stack. To manipulate partitions efficiently, we represent them as lists of trees (forests). Each equivalence class \( C \) in the partition \( \Pi \) is represented by a tree structure consisting of nodes and parent pointers, with the pointers from the sons of a node to the node itself. The root has a null pointer. Each node also maintains a counter keeping track of the number of nodes in the subtree rooted at that node.

Note that pointers can be avoided. We can represent a forest of \( n \) nodes as a list of \( n \) pairs of the form \((\text{father}, \text{count})\). If \((\text{father}, \text{count})\) is the \( i \)th pair in the list, then \( \text{father} = 0 \) iff node \( i \) is a root node, otherwise, \( \text{father} \) is the index of the node in the list which is the
2.9. A FAST ALGORITHM FOR CHECKING STATE EQUIVALENCE

parent of node \(i\). The number \(\text{count}\) is the total number of nodes in the tree rooted at the \(i\)th node.

For example, the following list of nine nodes

\[((0, 3), (0, 2), (1, 1), (0, 2), (0, 2), (1, 1), (2, 1), (4, 1), (5, 1))\]

represents a forest consisting of the following four trees:

![Figure 2.18: A forest of four trees](image)

Two functions \(\text{union}\) and \(\text{find}\) are defined as follows. Given a state \(p\), \(\text{find}(p, \Pi)\) finds the root of the tree containing \(p\) as a node (not necessarily a leaf). Given two root nodes \(p, q\), \(\text{union}(p, q, \Pi)\) forms a new partition by merging the two trees with roots \(p\) and \(q\) as follows: if the counter of \(p\) is smaller than that of \(q\), then let the root of \(p\) point to \(q\), else let the root of \(q\) point to \(p\).

For example, given the two trees shown on the left in Figure 2.19, \(\text{find}(6, \Pi)\) returns 3 and \(\text{find}(8, \Pi)\) returns 4. Then \(\text{union}(3, 4, \Pi)\) yields the tree shown on the right in Figure 2.19.

![Figure 2.19: Applying the function \text{union} to the trees rooted at 3 and 4](image)

In order to speed up the algorithm, using an idea due to Tarjan, we can modify \(\text{find}\) as follows: during a call \(\text{find}(p, \Pi)\), as we follow the path from \(p\) to the root \(r\) of the tree containing \(p\), we redirect the parent pointer of every node \(q\) on the path from \(p\) (including \(p\) itself) to \(r\) (we perform path compression). For example, applying \(\text{find}(8, \Pi)\) to the tree shown on the right in Figure 2.19 yields the tree shown in Figure 2.20.
Figure 2.20: The result of applying \textit{find} with path compression

Then, the algorithm is as follows:
2.9. A FAST ALGORITHM FOR CHECKING STATE EQUIVALENCE

function unif\[p, q, \Pi, dd\]: flag;
begin
trans := left\(dd\); ff := right\(dd\); pq := (p, q); st := (pq); flag := 1;
k := Length(first\(trans\));
while st \neq () \land flag \neq 0 do
uv := top\(st\); uu := left\(uv\); vv := right\(uv\);
pop\(st\);
if bad\(ff, uv\) = 1 then flag := 0
else
u := find\(uu, \Pi\); v := find\(vv, \Pi\);
if u \neq v then
union\(u, v, \Pi\);
for i = 1 to k do
u1 := delta\(trans, uu, k - i + 1\); v1 := delta\(trans, vv, k - i + 1\);
vw := (u1, v1); push\(st, uv\)
endfor
endif
endif
endwhile
end

The initial partition \(\Pi\) is the identity relation on \(Q\), i.e., it consists of blocks \(\{q\}\) for all states \(q \in Q\). The algorithm uses a stack \(st\). We are assuming that the DFA \(dd\) is specified by a list of two sublists, the first list, denoted \(left(dd)\) in the pseudo-code above, being a representation of the transition function, and the second one, denoted \(right(dd)\), the set of final states. The transition function itself is a list of lists, where the \(i\)-th list represents the \(i\)-th row of the transition table for \(dd\). The function \(delta\) is such that \(delta(trans,i,j)\) returns the \(j\)-th state in the \(i\)-th row of the transition table of \(dd\). For example, we have the DFA

\[dd = ((2, 3), (2, 4), (2, 3), (2, 5), (2, 3), (7, 6), (7, 8), (7, 9), (7, 6)), (5, 9))\]

consisting of 9 states labeled 1, \ldots, 9, and two final states 5 and 9 shown in Figure 2.21. Also, the alphabet has two letters, since every row in the transition table consists of two entries. For example, the two transitions from state 3 are given by the pair (2, 3), which indicates that \(\delta(3, a) = 2\) and \(\delta(3, b) = 3\).

The sequence of steps performed by the algorithm starting with \(p = 1\) and \(q = 6\) is shown below. At every step, we show the current pair of states, the partition, and the stack.
Figure 2.21: Testing state equivalence in a DFA

\[ p = 1, q = 6, \Pi = \{\{1, 6\}, \{2\}, \{3\}, \{4\}, \{5\}, \{7\}, \{8\}, \{9\}\}, \text{st} = \{\{1, 6\}\} \]

Figure 2.22: Testing state equivalence in a DFA

\[ p = 2, q = 7, \Pi = \{\{1, 6\}, \{2, 7\}, \{3\}, \{4\}, \{5\}, \{8\}, \{9\}\}, \text{st} = \{\{3, 6\}, \{2, 7\}\} \]
2.9. A FAST ALGORITHM FOR CHECKING STATE EQUIVALENCE

Figure 2.23: Testing state equivalence in a DFA

$p = 4, q = 8, \Pi = \{\{1, 6\}, \{2, 7\}, \{3\}, \{4, 8\}, \{5\}, \{9\}\}, \text{st} = \{\{3, 6\}, \{4, 8\}\}$

Figure 2.24: Testing state equivalence in a DFA

$p = 5, q = 9, \Pi = \{\{1, 6\}, \{2, 7\}, \{3\}, \{4, 8\}, \{5, 9\}\}, \text{st} = \{\{3, 6\}, \{5, 9\}\}$
Figure 2.25: Testing state equivalence in a DFA

\[ p = 3, q = 6, \Pi = \{\{1, 3, 6\}, \{2, 7\}, \{4, 8\}, \{5, 9\}\}, st = \{\{3, 6\}, \{3, 6\}\} \]

Since states 3 and 6 belong to the first block of the partition, the algorithm terminates.

Since no block of the partition contains a bad pair, the states \( p = 1 \) and \( q = 6 \) are equivalent.

Let us now test whether the states \( p = 3 \) and \( q = 7 \) are equivalent.

Figure 2.26: Testing state equivalence in a DFA
\[ p = 3, q = 7, \; \Pi = \{\{1\}, \{2\}, \{3, 7\}, \{4\}, \{5\}, \{6\}, \{8\}, \{9\}\}, \; st = \{\{3, 7\}\} \]

Figure 2.27: Testing state equivalence in a DFA

\[ p = 2, q = 7, \; \Pi = \{\{1\}, \{2, 3, 7\}, \{4\}, \{5\}, \{6\}, \{8\}, \{9\}\}, \; st = \{\{3, 8\}, \{2, 7\}\} \]

Figure 2.28: Testing state equivalence in a DFA
CHAPTER 2. REGULAR LANGUAGES, MINIMIZATION OF DFA'S

\[ p = 4, q = 8, \ \Pi = \{\{1\}, \{2, 3, 7\}, \{4, 8\}, \{5\}, \{6\}, \{9\}\}, \ st = \{\{3, 8\}, \{4, 8\}\} \]

![Figure 2.29: Testing state equivalence in a DFA](image)

\[ p = 5, q = 9, \ \Pi = \{\{1\}, \{2, 3, 7\}, \{4, 8\}, \{5, 9\}, \{6\}\}, \ st = \{\{3, 8\}, \{5, 9\}\} \]

![Figure 2.30: Testing state equivalence in a DFA](image)
\[ p = 3, q = 6, \; \Pi = \{\{1\}, \{2, 3, 6, 7\}, \{4, 8\}, \{5, 9\}\}, \; st = \{\{3, 8\}, \{3, 6\}\} \]

Figure 2.31: Testing state equivalence in a DFA

\[ p = 3, q = 8, \; \Pi = \{\{1\}, \{2, 3, 4, 6, 7, 8\}, \{5, 9\}\}, \; st = \{\{3, 8\}\} \]

Figure 2.32: Testing state equivalence in a DFA
$p = 3, q = 9, \Pi = \{\{1\}, \{2, 3, 4, 6, 7, 8\}, \{5, 9\}\}, st = \{\{3, 9\}\}$

Since the pair $(3, 9)$ is a bad pair, the algorithm stops, and the states $p = 3$ and $q = 7$ are inequivalent.
Chapter 3

Context-Free Grammars,
Context-Free Languages, Parse Trees
and Ogden’s Lemma

3.1 Context-Free Grammars

A context-free grammar basically consists of a finite set of grammar rules. In order to define grammar rules, we assume that we have two kinds of symbols: the terminals, which are the symbols of the alphabet underlying the languages under consideration, and the nonterminals, which behave like variables ranging over strings of terminals. A rule is of the form $A \rightarrow \alpha$, where $A$ is a single nonterminal, and the right-hand side $\alpha$ is a string of terminal and/or nonterminal symbols. As usual, first we need to define what the object is (a context-free grammar), and then we need to explain how it is used. Unlike automata, grammars are used to generate strings, rather than recognize strings.

Definition 3.1. A context-free grammar (for short, CFG) is a quadruple $G = (V, \Sigma, P, S)$, where

- $V$ is a finite set of symbols called the vocabulary (or set of grammar symbols);
- $\Sigma \subseteq V$ is the set of terminal symbols (for short, terminals);
- $S \in (V - \Sigma)$ is a designated symbol called the start symbol;
- $P \subseteq (V - \Sigma) \times V^*$ is a finite set of productions (or rewrite rules, or rules).

The set $N = V - \Sigma$ is called the set of nonterminal symbols (for short, nonterminals). Thus, $P \subseteq N \times V^*$, and every production $\langle A, \alpha \rangle$ is also denoted as $A \rightarrow \alpha$. A production of the form $A \rightarrow \epsilon$ is called an epsilon rule, or null rule.
Remark: Context-free grammars are sometimes defined as $G = (V_N, V_T, P, S)$. The correspondence with our definition is that $\Sigma = V_T$ and $N = V_N$, so that $V = V_N \cup V_T$. Thus, in this other definition, it is necessary to assume that $V_T \cap V_N = \emptyset$.

Example 1. $G_1 = (\{E, a, b\}, \{a, b\}, P, E)$, where $P$ is the set of rules

$$
E \rightarrow aEb,
E \rightarrow ab.
$$

As we will see shortly, this grammar generates the language $L_1 = \{a^n b^n \mid n \geq 1\}$, which is not regular.

Example 2. $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where $P$ is the set of rules

$$
E \rightarrow E + E,
E \rightarrow E * E,
E \rightarrow (E),
E \rightarrow a.
$$

This grammar generates a set of arithmetic expressions.

### 3.2 Derivations and Context-Free Languages

The productions of a grammar are used to derive strings. In this process, the productions are used as rewrite rules. Formally, we define the derivation relation associated with a context-free grammar. First, let us review the concepts of transitive closure and reflexive and transitive closure of a binary relation.

Given a set $A$, a binary relation $R$ on $A$ is any set of ordered pairs, i.e. $R \subseteq A \times A$. For short, instead of binary relation, we often simply say relation. Given any two relations $R, S$ on $A$, their composition $R \circ S$ is defined as

$$
R \circ S = \{(x, y) \in A \times A \mid \exists z \in A, (x, z) \in R \text{ and } (z, y) \in S\}.
$$

The identity relation $I_A$ on $A$ is the relation $I_A$ defined such that

$$
I_A = \{(x, x) \mid x \in A\}.
$$

For short, we often denote $I_A$ as $I$. Note that

$$
R \circ I = I \circ R = R
$$

for every relation $R$ on $A$. Given a relation $R$ on $A$, for any $n \geq 0$ we define $R^n$ as follows:

$$
R^0 = I,
R^{n+1} = R^n \circ R.
$$
It is obvious that $R^1 = R$. It is also easily verified by induction that $R^n \circ R = R \circ R^n$. The *transitive closure* $R^+$ of the relation $R$ is defined as

$$R^+ = \bigcup_{n \geq 1} R^n.$$ 

It is easily verified that $R^+$ is the smallest transitive relation containing $R$, and that $(x, y) \in R^+$ iff there is some $n \geq 1$ and some $x_0, x_1, \ldots, x_n \in A$ such that $x_0 = x$, $x_n = y$, and $(x_i, x_{i+1}) \in R$ for all $i$, $0 \leq i \leq n - 1$. The *transitive and reflexive closure* $R^*$ of the relation $R$ is defined as

$$R^* = \bigcup_{n \geq 0} R^n.$$ 

Clearly, $R^* = R^+ \cup I$. It is easily verified that $R^*$ is the smallest transitive and reflexive relation containing $R$.

**Definition 3.2.** Given a context-free grammar $G = (V, \Sigma, P, S)$, the (one-step) *derivation relation* $\Rightarrow_G$ associated with $G$ is the binary relation $\Rightarrow_G \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \Rightarrow_G \beta$$

iff there exist $\lambda, \rho \in V^*$, and some production $(A \rightarrow \gamma) \in P$, such that

$$\alpha = \lambda A \rho \quad \text{and} \quad \beta = \lambda \gamma \rho.$$ 

The transitive closure of $\Rightarrow_G$ is denoted as $\Rightarrow^+_G$ and the reflexive and transitive closure of $\Rightarrow_G$ is denoted as $\Rightarrow^*_G$.

When the grammar $G$ is clear from the context, we usually omit the subscript $G$ in $\Rightarrow_G$, $\Rightarrow^+_G$, and $\Rightarrow^*_G$.

A string $\alpha \in V^*$ such that $S \Rightarrow^* \alpha$ is called a *sentential form*, and a string $w \in \Sigma^*$ such that $S \Rightarrow^* w$ is called a *sentence*. A derivation $\alpha \Rightarrow^*_n \beta$ involving $n$ steps is denoted as $\alpha \Rightarrow^n \beta$.

Note that a derivation step

$$\alpha \Rightarrow_G \beta$$

is rather nondeterministic. Indeed, one can choose among various occurrences of nonterminals $A$ in $\alpha$, and also among various productions $A \rightarrow \gamma$ with left-hand side $A$.

For example, using the grammar $G_1 = (\{E, a, b\}, \{a, b\}, P, E)$, where $P$ is the set of rules

- $E \rightarrow aEb$,
- $E \rightarrow ab$, 
every derivation from $E$ is of the form

$$E \Rightarrow a^nEb^n \Rightarrow a^nabb^n = a^{n+1}b^{n+1},$$

or

$$E \Rightarrow a^nEb^n \Rightarrow a^nabEb^n = a^{n+1}b^{n+1},$$

where $n \geq 0$.

Grammar $G_1$ is very simple: every string $a^n b^n$ has a unique derivation. This is usually not the case. For example, using the grammar $G_2 = (\{E, +, *, (,), a\}, \{+, *, (,), a\}, P, E)$, where $P$ is the set of rules

$$
\begin{align*}
E &\rightarrow E + E, \\
E &\rightarrow E * E, \\
E &\rightarrow (E), \\
E &\rightarrow a,
\end{align*}
$$

the string $a + a * a$ has the following distinct derivations, where the boldface indicates which occurrence of $E$ is rewritten:

$$
\begin{align*}
E \rightarrow E \ast E \Rightarrow E + E \ast E \\
\Rightarrow a + E \ast E \Rightarrow a + a \ast E \Rightarrow a + a \ast a,
\end{align*}
$$

and

$$
\begin{align*}
E \rightarrow E + E \Rightarrow a + E \\
\Rightarrow a + E \ast E \Rightarrow a + a \ast E \Rightarrow a + a \ast a.
\end{align*}
$$

In the above derivations, the leftmost occurrence of a nonterminal is chosen at each step. Such derivations are called leftmost derivations. We could systematically rewrite the rightmost occurrence of a nonterminal, getting rightmost derivations. The string $a + a * a$ also has the following two rightmost derivations, where the boldface indicates which occurrence of $E$ is rewritten:

$$
\begin{align*}
E \rightarrow E + E \Rightarrow E + E \ast E \\
\Rightarrow E + E \ast a \Rightarrow E + a * a \Rightarrow a + a \ast a,
\end{align*}
$$

and

$$
\begin{align*}
E \rightarrow E \ast E \Rightarrow E \ast a \\
\Rightarrow E + E \ast a \Rightarrow E + a * a \Rightarrow a + a \ast a.
\end{align*}
$$

The language generated by a context-free grammar is defined as follows.
Definition 3.3. Given a context-free grammar $G = (V, \Sigma, P, S)$, the *language generated by* $G$ is the set

$$L(G) = \{ w \in \Sigma^* \mid S \xrightarrow{+} w \}.$$ 

A language $L \subseteq \Sigma^*$ is a *context-free language* (for short, CFL) iff $L = L(G)$ for some context-free grammar $G$.

It is technically very useful to consider derivations in which the leftmost nonterminal is always selected for rewriting, and dually, derivations in which the rightmost nonterminal is always selected for rewriting.

Definition 3.4. Given a context-free grammar $G = (V, \Sigma, P, S)$, the *one-step leftmost derivation relation* $\xrightarrow{lm}$ associated with $G$ is the binary relation $\xrightarrow{lm} \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \xrightarrow{lm} \beta$$

iff there exist $u \in \Sigma^*$, $\rho \in V^*$, and some production $(A \rightarrow \gamma) \in P$, such that

$$\alpha = uA\rho \quad \text{and} \quad \beta = u\gamma\rho.$$ 

The transitive closure of $\xrightarrow{lm}$ is denoted as $\xrightarrow{+}_{lm}$ and the reflexive and transitive closure of $\xrightarrow{lm}$ is denoted as $\xrightarrow{*}_{lm}$. The *one-step rightmost derivation relation* $\xrightarrow{rm}$ associated with $G$ is the binary relation $\xrightarrow{rm} \subseteq V^* \times V^*$ defined as follows: for all $\alpha, \beta \in V^*$, we have

$$\alpha \xrightarrow{rm} \beta$$

iff there exist $\lambda \in V^*$, $v \in \Sigma^*$, and some production $(A \rightarrow \gamma) \in P$, such that

$$\alpha = \lambda Av \quad \text{and} \quad \beta = \lambda\gamma v.$$ 

The transitive closure of $\xrightarrow{rm}$ is denoted as $\xrightarrow{+}_{rm}$ and the reflexive and transitive closure of $\xrightarrow{rm}$ is denoted as $\xrightarrow{*}_{rm}$.

Remarks: It is customary to use the symbols $a, b, c, d, e$ for terminal symbols, and the symbols $A, B, C, D, E$ for nonterminal symbols. The symbols $u, v, w, x, y, z$ denote terminal strings, and the symbols $\alpha, \beta, \gamma, \lambda, \rho, \mu$ denote strings in $V^*$. The symbols $X, Y, Z$ usually denote symbols in $V$.

Given a context-free grammar $G = (V, \Sigma, P, S)$, *parsing* a string $w$ consists in finding out whether $w \in L(G)$, and if so, in producing a derivation for $w$. The following lemma is technically very important. It shows that leftmost and rightmost derivations are “universal”. This has some important practical implications for the complexity of parsing algorithms.
Lemma 3.1. Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For every $w \in \Sigma^*$, for every derivation $S \xrightarrow{+} w$, there is a leftmost derivation $S \xrightarrow{\text{lm}} w$, and there is a rightmost derivation $S \xrightarrow{\text{rm}} w$.

Proof. Of course, we have to somehow use induction on derivations, but this is a little tricky, and it is necessary to prove a stronger fact. We treat leftmost derivations, rightmost derivations being handled in a similar way.

Claim: For every $w \in \Sigma^*$, for every $\alpha \in V^+$, for every $n \geq 1$, if $\alpha \xrightarrow{n} w$, then there is a leftmost derivation $\alpha \xrightarrow{\text{lm}} w$.

The claim is proved by induction on $n$.

For $n = 1$, there exist some $\lambda, \rho \in V^*$ and some production $A \rightarrow \gamma$, such that $\alpha = \lambda A \rho$ and $w = \lambda \gamma \rho$. Since $w$ is a terminal string, $\lambda, \rho$, and $\gamma$, are terminal strings. Thus, $A$ is the only nonterminal in $\alpha$, and the derivation step $\alpha \xrightarrow{1} w$ is a leftmost step (and a rightmost step!).

If $n > 1$, then the derivation $\alpha \xrightarrow{n} w$ is of the form

$\alpha \xrightarrow{n-1} w$.

There are two subcases.

Case 1. If the derivation step $\alpha \xrightarrow{n-1} w$ is a leftmost step $\alpha \xrightarrow{\text{lm}} w$, by the induction hypothesis, there is a leftmost derivation $\alpha \xrightarrow{\text{lm}} w$, and we get the leftmost derivation

$\alpha \xrightarrow{\text{lm}} \alpha_1 \xrightarrow{\text{lm}} w$.

Case 2. The derivation step $\alpha \xrightarrow{n-1} w$ is a not a leftmost step. In this case, there must be some $u \in \Sigma^*$, $\mu, \rho \in V^*$, some nonterminals $A$ and $B$, and some production $B \rightarrow \delta$, such that

$\alpha = uAB \rho$ and $\alpha_1 = uA\mu \delta \rho$,

where $A$ is the leftmost nonterminal in $\alpha$. Since we have a derivation $\alpha_1 \xrightarrow{n-1} w$ of length $n - 1$, by the induction hypothesis, there is a leftmost derivation

$\alpha_1 \xrightarrow{\text{lm}} w$.

Since $\alpha_1 = uA\mu \delta \rho$ where $A$ is the leftmost terminal in $\alpha_1$, the first step in the leftmost derivation $\alpha_1 \xrightarrow{\text{lm}} w$ is of the form

$uA\mu \delta \rho \xrightarrow{\text{lm}} w \gamma \mu \delta \rho$.
3.2. DERIVATIONS AND CONTEXT-FREE LANGUAGES

for some production $A \rightarrow \gamma$. Thus, we have a derivation of the form

$$\alpha = uA\mu B\rho \overset{\text{lm}}{\Rightarrow} uA\mu \delta\rho \overset{n-2}{\Rightarrow} \frac{\text{lm}}{\text{lm}} \frac{\text{lm}}{\text{lm}} w.$$ 

We can commute the first two steps involving the productions $B \rightarrow \delta$ and $A \rightarrow \gamma$, and we get the derivation

$$\alpha = uA\mu B\rho \overset{\text{lm}}{\Rightarrow} u\gamma \mu B\rho \overset{n-2}{\Rightarrow} \frac{\text{lm}}{\text{lm}} \frac{\text{lm}}{\text{lm}} w.$$ 

This may no longer be a leftmost derivation, but the first step is leftmost, and we are back in case 1. Thus, we conclude by applying the induction hypothesis to the derivation $u\gamma B\rho \overset{n-1}{\Rightarrow} w$, as in case 1.

Lemma 3.1 implies that

$$L(G) = \{ w \in \Sigma^* \mid S \overset{+}{\text{lm}} \Rightarrow w \} = \{ w \in \Sigma^* \mid S \overset{+}{\text{rm}} \Rightarrow w \}.$$ 

We observed that if we consider the grammar $G_2 = (\{E, +, \ast, (, )\}, \{+, \ast, (, )\}, P, E)$, where $P$ is the set of rules

- $E \rightarrow E + E$,
- $E \rightarrow E \ast E$,
- $E \rightarrow (E)$,
- $E \rightarrow a$,

the string $a + a \ast a$ has the following two distinct leftmost derivations, where the boldface indicates which occurrence of $E$ is rewritten:

$$E \Rightarrow E \ast E \Rightarrow E + E \ast E$$
$$\Rightarrow a + E \ast E \Rightarrow a + a \ast E \Rightarrow a + a \ast a,$$

and

$$E \Rightarrow E + E \Rightarrow a + E$$
$$\Rightarrow a + E \ast E \Rightarrow a + a \ast E \Rightarrow a + a \ast a.$$ 

When this happens, we say that we have an ambiguous grammars. In some cases, it is possible to modify a grammar to make it unambiguous. For example, the grammar $G_2$ can be modified as follows.

Let $G_3 = (\{E, T, F, +, \ast, (, )\}, \{+, \ast, (, )\}, P, E)$, where $P$ is the set of rules

- $E \rightarrow E + T$,
- $E \rightarrow T$,
- $T \rightarrow T \ast F$,
- $T \rightarrow F$,
- $F \rightarrow (E)$,
- $F \rightarrow a$. 

We leave as an exercise to show that $L(G_3) = L(G_2)$, and that every string in $L(G_3)$ has a unique leftmost derivation. Unfortunately, it is not always possible to modify a context-free grammar to make it unambiguous. There exist context-free languages that have no unambiguous context-free grammars. For example, the language

$$L_3 = \{a^m b^m c^n \mid m,n \geq 1\} \cup \{a^m b^n c^n \mid m,n \geq 1\}$$

is context-free, since it is generated by the following context-free grammar:

- $S \rightarrow S_1,$
- $S \rightarrow S_2,$
- $S_1 \rightarrow XC,$
- $S_2 \rightarrow AY,$
- $X \rightarrow aXb,$
- $X \rightarrow ab,$
- $Y \rightarrow bYc,$
- $Y \rightarrow bc,$
- $A \rightarrow aA,$
- $A \rightarrow a,$
- $C \rightarrow cC,$
- $C \rightarrow c.$

However, it can be shown that $L_3$ has no unambiguous grammars. All this motivates the following definition.

**Definition 3.5.** A context-free grammar $G = (V, \Sigma, P, S)$ is *ambiguous* if there is some string $w \in L(G)$ that has two distinct leftmost derivations (or two distinct rightmost derivations). Thus, a grammar $G$ is *unambiguous* if every string $w \in L(G)$ has a unique leftmost derivation (or a unique rightmost derivation). A context-free language $L$ is *inherently ambiguous* if every CFG $G$ for $L$ is ambiguous.

Whether or not a grammar is ambiguous affects the complexity of parsing. Parsing algorithms for unambiguous grammars are more efficient than parsing algorithms for ambiguous grammars.

We now consider various normal forms for context-free grammars.

### 3.3 Normal Forms for Context-Free Grammars, Chomsky Normal Form

One of the main goals of this section is to show that every CFG $G$ can be converted to an equivalent grammar in *Chomsky Normal Form (for short, CNF)*. A context-free grammar
3.3. NORMAL FORMS FOR CONTEXT-FREE GRAMMARS

$G = (V, \Sigma, P, S)$ is in Chomsky Normal Form if its productions are of the form

\[
\begin{align*}
A & \rightarrow BC, \\
A & \rightarrow a, \text{ or} \\
S & \rightarrow \epsilon,
\end{align*}
\]

where $A, B, C \in N$, $a \in \Sigma$, $S \rightarrow \epsilon$ is in $P$ iff $\epsilon \in L(G)$, and $S$ does not occur on the right-hand side of any production.

Note that a grammar in Chomsky Normal Form does not have $\epsilon$-rules, i.e., rules of the form $A \rightarrow \epsilon$, except when $\epsilon \in L(G)$, in which case $S \rightarrow \epsilon$ is the only $\epsilon$-rule. It also does not have chain rules, i.e., rules of the form $A \rightarrow B$, where $A, B \in N$. Thus, in order to convert a grammar to Chomsky Normal Form, we need to show how to eliminate $\epsilon$-rules and chain rules. This is not the end of the story, since we may still have rules of the form $A \rightarrow \alpha$ where either $|\alpha| \geq 3$ or $|\alpha| \geq 2$ and $\alpha$ contains terminals. However, dealing with such rules is a simple recoding matter, and we first focus on the elimination of $\epsilon$-rules and chain rules. It turns out that $\epsilon$-rules must be eliminated first.

The first step to eliminate $\epsilon$-rules is to compute the set $E(G)$ of erasable (or nullable) nonterminals

\[
E(G) = \{A \in N \mid A \xrightarrow{+} \epsilon\}.
\]

The set $E(G)$ is computed using a sequence of approximations $E_i$ defined as follows:

\[
\begin{align*}
E_0 & = \{A \in N \mid (A \rightarrow \epsilon) \in P\}, \\
E_{i+1} & = E_i \cup \{A \mid \exists(B \rightarrow B_1 \ldots B_j B_{j+1} \ldots B_k) \in P, B_j \in E_i, 1 \leq j \leq k\}.
\end{align*}
\]

Clearly, the $E_i$ form an ascending chain

\[
E_0 \subseteq E_1 \subseteq \cdots \subseteq E_i \subseteq E_{i+1} \subseteq \cdots \subseteq N,
\]

and since $N$ is finite, there is a least $i$, say $i_0$, such that $E_{i_0} = E_{i_0+1}$. We claim that $E(G) = E_{i_0}$. Actually, we prove the following lemma.

**Lemma 3.2.** Given a context-free grammar $G = (V, \Sigma, P, S)$, one can construct a context-free grammar $G' = (V', \Sigma, P', S')$ such that:

1. $L(G') = L(G)$;  
2. $P'$ contains no $\epsilon$-rules other than $S' \rightarrow \epsilon$, and $S' \rightarrow \epsilon \in P'$ iff $\epsilon \in L(G)$;  
3. $S'$ does not occur on the right-hand side of any production in $P'$.

**Proof.** We begin by proving that $E(G) = E_{i_0}$. For this, we prove that $E(G) \subseteq E_{i_0}$ and $E_{i_0} \subseteq E(G)$.

To prove that $E_{i_0} \subseteq E(G)$, we proceed by induction on $i$. Since $E_0 = \{A \in N \mid (A \rightarrow \epsilon) \in P\}$, we have $A \xrightarrow{+} \epsilon$, and thus $A \in E(G)$. By the induction hypothesis, $E_i \subseteq$
If $A \in E_{i+1}$, either $A \in E_i$ and then $A \in E(G)$, or there is some production $(A \to B_1 \ldots B_j \ldots B_k) \in P$, such that $B_j \in E_i$ for all $j, 1 \leq j \leq k$. By the induction hypothesis, $B_j \Rightarrow^* \epsilon$ for each $j, 1 \leq j \leq k$, and thus

$$A \Rightarrow B_1 \ldots B_j \ldots B_k \Rightarrow B_2 \ldots B_j \ldots B_k \Rightarrow B_j \ldots B_k \Rightarrow \epsilon,$$

which shows that $A \in E(G)$.

To prove that $E(G) \subseteq E_{i_0}$, we also proceed by induction, but on the length of a derivation $A \Rightarrow^* \epsilon$. If $A \Rightarrow^* \epsilon$, then $A \Rightarrow \epsilon \in P$, and thus $A \in E_0$ since $E_0 = \{ A \in N \mid (A \to \epsilon) \in P \}$. If $A \Rightarrow^{n+1} \epsilon$, then

$$A \Rightarrow \alpha \Rightarrow^n \epsilon,$$

for some production $A \to \alpha \in P$. If $\alpha$ contains terminals of nonterminals not in $E(G)$, it is impossible to derive $\epsilon$ from $\alpha$, and thus, we must have $\alpha = B_1 \ldots B_j \ldots B_k$, with $B_j \in E(G)$, for all $j, 1 \leq j \leq k$. However, $B_j \Rightarrow^n \epsilon$ where $n_j \leq n$, and by the induction hypothesis, $B_j \in E_{i_0}$. But then, we get $A \in E_{i_0+1} = E_{i_0}$, as desired.

Having shown that $E(G) = E_{i_0}$, we construct the grammar $G'$. Its set of productions $P'$ is defined as follows. First, we create the production $S' \to S$ where $S' \notin V$, to make sure that $S'$ does not occur on the right-hand side of any rule in $P'$. Let

$$P_1 = \{ A \to \alpha \in P \mid \alpha \in V^+ \} \cup \{ S' \to S \},$$

and let $P_2$ be the set of productions

$$P_2 = \{ A \to \alpha_1 \alpha_2 \ldots \alpha_k \alpha_{k+1} \mid \exists \alpha_1 \in V^*, \ldots, \exists \alpha_{k+1} \in V^*, \exists B_1 \in E(G), \ldots, \exists B_k \in E(G) \}
A \to \alpha_1 B_1 \alpha_2 \ldots \alpha_k B_k \alpha_{k+1} \in P, \ k \geq 1, \ \alpha_1 \ldots \alpha_{k+1} \neq \epsilon \}.$$

Note that $\epsilon \in L(G)$ iff $S \in E(G)$. If $S \notin E(G)$, then let $P' = P_1 \cup P_2$, and if $S \in E(G)$, then let $P' = P_1 \cup P_2 \cup \{ S' \to \epsilon \}$. We claim that $L(G') = L(G)$, which is proved by showing that every derivation using $G$ can be simulated by a derivation using $G'$, and vice-versa. All the conditions of the lemma are now met.

From a practical point of view, the construction or lemma 3.2 is very costly. For example, given a grammar containing the productions

$$S \to ABCDEF,$$
$$A \to \epsilon,$$
$$B \to \epsilon,$$
$$C \to \epsilon,$$
$$D \to \epsilon,$$
$$E \to \epsilon,$$
$$F \to \epsilon,$$
$$\ldots \to \ldots,$$
3.3. NORMAL FORMS FOR CONTEXT-FREE GRAMMARS

eliminating \( \epsilon \)-rules will create \( 2^6 - 1 = 63 \) new rules corresponding to the 63 nonempty subsets of the set \( \{A, B, C, D, E, F\} \). We now turn to the elimination of chain rules.

It turns out that matters are greatly simplified if we first apply lemma 3.2 to the input grammar \( G \), and we explain the construction assuming that \( G = (V, \Sigma, P, S) \) satisfies the conditions of lemma 3.2. For every nonterminal \( A \in N \), we define the set
\[
I_A = \{ B \in N \mid A \xrightarrow{\epsilon} B \}.
\]
The sets \( I_A \) are computed using approximations \( I_{A,i} \) defined as follows:
\[
I_{A,0} = \{ B \in N \mid (A \to B) \in P \},
I_{A,i+1} = I_{A,i} \cup \{ C \in N \mid \exists (B \to C) \in P, \text{ and } B \in I_{A,i} \}.
\]
Clearly, for every \( A \in N \), the \( I_{A,i} \) form an ascending chain
\[
I_{A,0} \subseteq I_{A,1} \subseteq \cdots \subseteq I_{A,i} \subseteq I_{A,i+1} \subseteq \cdots \subseteq N,
\]
and since \( N \) is finite, there is a least \( i \), say \( i_0 \), such that \( I_{A,i_0} = I_{A,i_0+1} \). We claim that \( I_A = I_{A,i_0} \). Actually, we prove the following lemma.

Lemma 3.3. Given a context-free grammar \( G = (V, \Sigma, P, S) \), one can construct a context-free grammar \( G' = (V', \Sigma, P', S') \) such that:

1. \( L(G') = L(G) \);
2. Every rule in \( P' \) is of the form \( A \to \alpha \) where \( |\alpha| \geq 2 \), or \( A \to a \) where \( a \in \Sigma \), or \( S' \to \epsilon \) iff \( \epsilon \in L(G) \);
3. \( S' \) does not occur on the right-hand side of any production in \( P' \).

Proof. First, we apply lemma 3.2 to the grammar \( G \), obtaining a grammar \( G_1 = (V_1, \Sigma, S_1, P_1) \). The proof that \( I_A = I_{A,i_0} \) is similar to the proof that \( E(G) = E_{i_0} \). First, we prove that \( I_{A,i} \subseteq I_A \) by induction on \( i \). This is straightforward. Next, we prove that \( I_A \subseteq I_{A,i_0} \) by induction on derivations of the form \( A \xrightarrow{\epsilon} B \). In this part of the proof, we use the fact that \( G_1 \) has no \( \epsilon \)-rules except perhaps \( S_1 \to \epsilon \), and that \( S_1 \) does not occur on the right-hand side of any rule. This implies that a derivation \( A \xrightarrow{n+1} C \) is necessarily of the form \( A \xrightarrow{n} B \xrightarrow{\epsilon} C \) for some \( B \in N \). Then, in the induction step, we have \( B \in I_{A,i_0} \), and thus \( C \in I_{A,i_0+1} = I_{A,i_0} \).

We now define the following sets of rules. Let
\[
P_2 = P_1 - \{ A \to B \mid A \to B \in P_1 \},
\]
and let
\[
P_3 = \{ A \to \alpha \mid B \to \alpha \in P_1, \alpha \notin N_1, \text{ and } B \in I_A \}.
\]
We claim that \( G' = (V_1, \Sigma, P_2 \cup P_3, S_1) \) satisfies the conditions of the lemma. For example, \( S_1 \) does not appear on the right-hand side of any production, since the productions in \( P_3 \) have right-hand sides from \( P_1 \), and \( S_1 \) does not appear on the right-hand side in \( P_1 \). It is also easily shown that \( L(G') = L(G_1) = L(G) \). \( \Box \)
Let us apply the method of lemma 3.3 to the grammar

\[ G_3 = (\{E, T, F, +, *, (, ), a\}, \{+, *, (, )\}, P, E), \]

where \( P \) is the set of rules

\[
\begin{align*}
E & \rightarrow E + T, \\
E & \rightarrow T, \\
T & \rightarrow T * F, \\
T & \rightarrow F, \\
F & \rightarrow (E), \\
F & \rightarrow a.
\end{align*}
\]

We get \( I_E = \{T, F\}, I_T = \{F\}, \text{ and } I_F = \emptyset \). The new grammar \( G'_3 \) has the set of rules

\[
\begin{align*}
E & \rightarrow E + T, \\
E & \rightarrow T * F, \\
E & \rightarrow (E), \\
E & \rightarrow a, \\
T & \rightarrow T * F, \\
T & \rightarrow (E), \\
T & \rightarrow a, \\
F & \rightarrow (E), \\
F & \rightarrow a.
\end{align*}
\]

At this stage, the grammar obtained in lemma 3.3 no longer has \( \epsilon \)-rules (except perhaps \( S' \rightarrow \epsilon \) iff \( \epsilon \in L(G) \)) or chain rules. However, it may contain rules \( A \rightarrow \alpha \) with \( |\alpha| \geq 3 \), or with \( |\alpha| \geq 2 \) and where \( \alpha \) contains terminals(s). To obtain the Chomsky Normal Form, we need to eliminate such rules. This is not difficult, but notationally a bit messy.

**Lemma 3.4.** Given a context-free grammar \( G = (V, \Sigma, P, S) \), one can construct a context-free grammar \( G' = (V', \Sigma, P', S') \) such that \( L(G') = L(G) \) and \( G' \) is in Chomsky Normal Form, that is, a grammar whose productions are of the form

\[
\begin{align*}
A & \rightarrow BC, \\
A & \rightarrow a, \quad \text{or} \\
S' & \rightarrow \epsilon,
\end{align*}
\]

where \( A, B, C \in N', a \in \Sigma, S' \rightarrow \epsilon \) is in \( P' \) iff \( \epsilon \in L(G) \), and \( S' \) does not occur on the right-hand side of any production in \( P' \).
Proof. First, we apply lemma 3.3, obtaining \( G_1 \). Let \( \Sigma_r \) be the set of terminals occurring on the right-hand side of rules \( A \to \alpha \in P_1 \), with \( |\alpha| \geq 2 \). For every \( a \in \Sigma_r \), let \( X_a \) be a new nonterminal not in \( V_1 \). Let

\[
P_2 = \{ X_a \to a \mid a \in \Sigma_r \}.
\]

Let \( P_{1,r} \) be the set of productions

\[
A \to \alpha_1 a_1 \alpha_2 \cdots \alpha_k a_k \alpha_{k+1},
\]

where \( a_1, \ldots, a_k \in \Sigma_r \) and \( \alpha_i \in N_1^* \). For every production

\[
A \to \alpha_1 a_1 \alpha_2 \cdots \alpha_k a_k \alpha_{k+1}
\]

in \( P_{1,r} \), let

\[
A \to \alpha_1 X_{a_1} \alpha_2 \cdots \alpha_k X_{a_k} \alpha_{k+1}
\]

be a new production, and let \( P_3 \) be the set of all such productions. Let \( P_4 = (P_1 - P_{1,r}) \cup P_2 \cup P_3 \). Now, productions \( A \to \alpha \in P_4 \) with \( |\alpha| \geq 2 \) do not contain terminals. However, we may still have productions \( A \to \alpha \in P_4 \) with \( |\alpha| \geq 3 \). We can perform some recoding using some new nonterminals. For every production of the form

\[
A \to B_1 \cdots B_k,
\]

where \( k \geq 3 \), create the new nonterminals

\[
[B_1 \cdots B_{k-1}], [B_1 \cdots B_{k-2}], \ldots, [B_1 B_2 B_3], [B_1 B_2],
\]

and the new productions

\[
A \to [B_1 \cdots B_{k-1}] B_k,
\]

\[
[B_1 \cdots B_{k-1}] \to [B_1 \cdots B_{k-2}] B_{k-1},
\]

\[
\cdots \to \cdots,
\]

\[
[B_1 B_2 B_3] \to [B_1 B_2] B_3,
\]

\[
[B_1 B_2] \to B_1 B_2.
\]

All the productions are now in Chomsky Normal Form, and it is clear that the same language is generated. \( \square \)
Applying the first phase of the method of lemma 3.4 to the grammar $G'_3$, we get the rules

\[
\begin{align*}
E & \rightarrow EX_+T, \\
E & \rightarrow TX_+F, \\
E & \rightarrow X(EX), \\
E & \rightarrow a, \\
T & \rightarrow TX_+F, \\
T & \rightarrow X(EX), \\
T & \rightarrow a, \\
F & \rightarrow X(EX), \\
F & \rightarrow a, \\
X_+ & \rightarrow +, \\
X_* & \rightarrow *, \\
X( & \rightarrow (, \\
X) & \rightarrow ).
\end{align*}
\]

After applying the second phase of the method, we get the following grammar in Chomsky Normal Form:

\[
\begin{align*}
E & \rightarrow [EX_+]T, \\
[EX_+] & \rightarrow EX_+, \\
E & \rightarrow [TX_+]F, \\
[TX_+] & \rightarrow TX_+, \\
E & \rightarrow [X(E)X], \\
[X(E)] & \rightarrow X(E), \\
E & \rightarrow a, \\
T & \rightarrow [TX_+]F, \\
T & \rightarrow [X(E)X], \\
T & \rightarrow a, \\
F & \rightarrow [X(E)X], \\
F & \rightarrow a, \\
X_+ & \rightarrow +, \\
X_* & \rightarrow *, \\
X( & \rightarrow (, \\
X) & \rightarrow ).
\end{align*}
\]

For large grammars, it is often convenient to use the abbreviation which consists in grouping productions having a common left-hand side, and listing the right-hand sides separated
3.4. **REGULAR LANGUAGES ARE CONTEXT-FREE**

by the symbol |. Thus, a group of productions

\[ A \rightarrow \alpha_1, \]
\[ A \rightarrow \alpha_2, \]
\[ \cdots \rightarrow \cdots, \]
\[ A \rightarrow \alpha_k, \]

may be abbreviated as

\[ A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_k. \]

An interesting corollary of the CNF is the following decidability result. There is an algorithm which, given a context-free grammar \( G \), given any string \( w \in \Sigma^* \), decides whether \( w \in L(G) \). Indeed, we first convert \( G \) to a grammar \( G' \) in Chomsky Normal Form. If \( w = \epsilon \), we can test whether \( \epsilon \in L(G) \), since this is the case iff \( S' \rightarrow \epsilon \in P' \). If \( w \neq \epsilon \), letting \( n = |w| \), note that since the rules are of the form \( A \rightarrow BC \) or \( A \rightarrow a \), where \( a \in \Sigma \), any derivation for \( w \) has \( n - 1 + n = 2n - 1 \) steps. Thus, we enumerate all (leftmost) derivations of length \( 2n - 1 \).

There are much better parsing algorithms than this naive algorithm. We now show that every regular language is context-free.

### 3.4 Regular Languages are Context-Free

The regular languages can be characterized in terms of very special kinds of context-free grammars, right-linear (and left-linear) context-free grammars.

**Definition 3.6.** A context-free grammar \( G = (V, \Sigma, P, S) \) is left-linear iff its productions are of the form

\[ A \rightarrow Ba, \]
\[ A \rightarrow a, \]
\[ A \rightarrow \epsilon. \]

where \( A, B \in N \), and \( a \in \Sigma \). A context-free grammar \( G = (V, \Sigma, P, S) \) is right-linear iff its productions are of the form

\[ A \rightarrow aB, \]
\[ A \rightarrow a, \]
\[ A \rightarrow \epsilon. \]

where \( A, B \in N \), and \( a \in \Sigma \).

The following lemma shows the equivalence between NFA’s and right-linear grammars.
Lemma 3.5. A language \( L \) is regular if and only if it is generated by some right-linear grammar.

Proof. Let \( L = L(D) \) for some DFA \( D = (Q, \Sigma, \delta, q_0, F) \). We construct a right-linear grammar \( G \) as follows. Let \( V = Q \cup \Sigma, S = q_0 \), and let \( P \) be defined as follows:

\[
P = \{ p \to aq \mid q = \delta(p, a), p, q \in Q, a \in \Sigma \} \cup \{ p \to \epsilon \mid p \in F \}.
\]

It is easily shown by induction on the length of \( w \) that

\[
p \xrightarrow{*} w \iff q = \delta^*(p, w),
\]

and thus, \( L(D) = L(G) \).

Conversely, let \( G = (V, \Sigma, P, S) \) be a right-linear grammar. First, let \( G' = (V', \Sigma, P', S) \) be the right-linear grammar obtained from \( G \) by adding the new nonterminal \( E \) to \( N \), replacing every rule in \( P \) of the form \( A \to a \) where \( a \in \Sigma \) by the rule \( A \to aE \), and adding the rule \( E \to \epsilon \). It is immediately verified that \( L(G') = L(G) \). Next, we construct the NFA \( M = (Q, \Sigma, \delta, q_0, F) \) as follows: \( Q = N' = N \cup \{ E \} \), \( q_0 = S \), \( F = \{ A \in N' \mid A \to \epsilon \} \), and

\[
\delta(A, a) = \{ B \in N' \mid A \to aB \in P' \},
\]

for all \( A \in N \) and all \( a \in \Sigma \). It is easily shown by induction on the length of \( w \) that

\[
A \xrightarrow{*} wB \iff B \in \delta^*(A, w),
\]

and thus, \( L(M) = L(G') = L(G) \). \( \square \)

A similar lemma holds for left-linear grammars. It is also easily shown that the regular languages are exactly the languages generated by context-free grammars whose rules are of the form

\[
A \to Bu,
\]

\[
A \to u,
\]

where \( A, B \in N \), and \( u \in \Sigma^* \).

3.5 Useless Productions in Context-Free Grammars

Given a context-free grammar \( G = (V, \Sigma, P, S) \), it may contain rules that are useless for a number of reasons. For example, consider the grammar \( G_3 = (\{ E, A, a, b \}, \{ a, b \}, P, E) \), where \( P \) is the set of rules:

\[
E \to aEb,
\]

\[
E \to ab,
\]

\[
E \to A,
\]

\[
A \to bAa.
\]
The problem is that the nonterminal $A$ does not derive any terminal strings, and thus, it is useless, as well as the last two productions. Let us now consider the grammar $G_4 = (\{E, A, a, b, c, d\}, \{a, b, c, d\}, P, E)$, where $P$ is the set of rules

\[
\begin{align*}
E & \rightarrow aEb, \\
E & \rightarrow ab, \\
A & \rightarrow cAd, \\
A & \rightarrow cd.
\end{align*}
\]

This time, the nonterminal $A$ generates strings of the form $c^nd^n$, but there is no derivation $E \Rightarrow^+ \alpha$ from $E$ where $A$ occurs in $\alpha$. The nonterminal $A$ is not connected to $E$, and the last two rules are useless. Fortunately, it is possible to find such useless rules, and to eliminate them.

Let $T(G)$ be the set of nonterminals that actually derive some terminal string, i.e.

\[T(G) = \{A \in (V - \Sigma) \mid \exists w \in \Sigma^*, \ A \Rightarrow^+ w\}.\]

The set $T(G)$ can be defined by stages. We define the sets $T_n \ (n \geq 1)$ as follows:

\[T_1 = \{A \in (V - \Sigma) \mid \exists (A \rightarrow w) \in P, \ \text{with} \ w \in \Sigma^*\},\]

and

\[T_{n+1} = T_n \cup \{A \in (V - \Sigma) \mid \exists (A \rightarrow \beta) \in P, \ \text{with} \ \beta \in (T_n \cup \Sigma)^*\}.\]

It is easy to prove that there is some least $n$ such that $T_{n+1} = T_n$, and that for this $n$, $T(G) = T_n$.

If $S \notin T(G)$, then $L(G) = \emptyset$, and $G$ is equivalent to the trivial grammar

\[G' = (\{S\}, \Sigma, \emptyset, S).\]

If $S \in T(G)$, then let $U(G)$ be the set of nonterminals that are actually useful, i.e.,

\[U(G) = \{A \in T(G) \mid \exists \alpha, \beta \in (T(G) \cup \Sigma)^*, \ S \Rightarrow^* \alpha A \beta\}.\]

The set $U(G)$ can also be computed by stages. We define the sets $U_n \ (n \geq 1)$ as follows:

\[U_1 = \{A \in T(G) \mid \exists (S \rightarrow \alpha A \beta) \in P, \ \text{with} \ \alpha, \beta \in (T(G) \cup \Sigma)^*\},\]

and

\[U_{n+1} = U_n \cup \{B \in T(G) \mid \exists (A \rightarrow \alpha B \beta) \in P, \ \text{with} \ A \in U_n, \ \alpha, \beta \in (T(G) \cup \Sigma)^*\}.\]

It is easy to prove that there is some least $n$ such that $U_{n+1} = U_n$, and that for this $n$, $U(G) = U_n \cup \{S\}$. Then, we can use $U(G)$ to transform $G$ into an equivalent CFG in
which every nonterminal is useful (i.e., for which \( V - \Sigma = U(G) \)). Indeed, simply delete all rules containing symbols not in \( U(G) \). The details are left as an exercise. We say that a context-free grammar \( G \) is reduced if all its nonterminals are useful, i.e., \( N = U(G) \).

It should be noted than although dull, the above considerations are important in practice. Certain algorithms for constructing parsers, for example, LR-parsers, may loop if useless rules are not eliminated!

We now consider another normal form for context-free grammars, the Greibach Normal Form.

### 3.6 The Greibach Normal Form

Every CFG \( G \) can also be converted to an equivalent grammar in Greibach Normal Form (shortly GNF). A context-free grammar \( G = (V, \Sigma, P, S) \) is in Greibach Normal Form iff its productions are of the form

\[
\begin{align*}
A & \rightarrow aBC, \\
A & \rightarrow aB, \\
A & \rightarrow a, \\
S & \rightarrow \epsilon,
\end{align*}
\]

where \( A, B, C \in N \), \( a \in \Sigma \), \( S \rightarrow \epsilon \) is in \( P \) iff \( \epsilon \in L(G) \), and \( S \) does not occur on the right-hand side of any production.

Note that a grammar in Greibach Normal Form does not have \( \epsilon \)-rules other than possibly \( S \rightarrow \epsilon \). More importantly, except for the special rule \( S \rightarrow \epsilon \), every rule produces some terminal symbol.

An important consequence of the Greibach Normal Form is that every nonterminal is not left recursive. A nonterminal \( A \) is left recursive iff \( A \xrightarrow{+} A\alpha \) for some \( \alpha \in V^* \). Left recursive nonterminals cause top-down deterministic parsers to loop. The Greibach Normal Form provides a way of avoiding this problem.

There are no easy proofs that every CFG can be converted to a Greibach Normal Form. A particularly elegant method due to Rosenkrantz using least fixed-points and matrices will be given in section 3.9.

**Lemma 3.6.** Given a context-free grammar \( G = (V, \Sigma, P, S) \), one can construct a context-free grammar \( G' = (V', \Sigma, P', S') \) such that \( L(G') = L(G) \) and \( G' \) is in Greibach Normal Form, that is, a grammar whose productions are of the form

\[
\begin{align*}
A & \rightarrow aBC, \\
A & \rightarrow aB, \\
A & \rightarrow a, \\
S' & \rightarrow \epsilon,
\end{align*}
\]
3.7 Least Fixed-Points

Context-free languages can also be characterized as least fixed-points of certain functions induced by grammars. This characterization yields a rather quick proof that every context-free grammar can be converted to Greibach Normal Form. This characterization also reveals very clearly the recursive nature of the context-free languages.

We begin by reviewing what we need from the theory of partially ordered sets.

**Definition 3.7.** Given a partially ordered set \( \langle A, \leq \rangle \), an \( \omega \)-chain \((a_n)_{n \geq 0}\) is a sequence such that \( a_n \leq a_{n+1} \) for all \( n \geq 0 \). The least-upper bound of an \( \omega \)-chain \((a_n)\) is an element \( a \in A \) such that:

1. \( a_n \leq a \), for all \( n \geq 0 \);
2. For any \( b \in A \), if \( a_n \leq b \), for all \( n \geq 0 \), then \( a \leq b \).

A partially ordered set \( \langle A, \leq \rangle \) is an \( \omega \)-chain complete poset iff it has a least element \( \bot \), and iff every \( \omega \)-chain has a least upper bound denoted as \( \bigsqcup a_n \).

**Remark:** The \( \omega \) in \( \omega \)-chain means that we are considering countable chains (\( \omega \) is the ordinal associated with the order-type of the set of natural numbers). This notation may seem arcane, but is standard in denotational semantics.

For example, given any set \( X \), the power set \( 2^X \) ordered by inclusion is an \( \omega \)-chain complete poset with least element \( \emptyset \). The Cartesian product \( \underbrace{2^X \times \cdots \times 2^X}_n \) ordered such that

\[
(A_1, \ldots, A_n) \leq (B_1, \ldots, B_n)
\]

iff \( A_i \subseteq B_i \) (where \( A_i, B_i \in 2^X \)) is an \( \omega \)-chain complete poset with least element \( (\emptyset, \ldots, \emptyset) \).

We are interested in functions between partially ordered sets.

**Definition 3.8.** Given any two partially ordered sets \( \langle A_1, \leq_1 \rangle \) and \( \langle A_2, \leq_2 \rangle \), a function \( f : A_1 \to A_2 \) is monotonic iff for all \( x, y \in A_1 \),

\[
x \leq_1 y \implies f(x) \leq_2 f(y).
\]

If \( \langle A_1, \leq_1 \rangle \) and \( \langle A_2, \leq_2 \rangle \) are \( \omega \)-chain complete posets, a function \( f : A_1 \to A_2 \) is \( \omega \)-continuous iff it is monotonic, and for every \( \omega \)-chain \((a_n)\),

\[
f(\bigsqcup a_n) = \bigsqcup f(a_n).
\]
Remark: Note that we are not requiring that an \( \omega \)-continuous function \( f : A_1 \to A_2 \) preserve least elements, i.e., it is possible that \( f(\bot_1) \neq \bot_2 \).

We now define the crucial concept of a least fixed-point.

**Definition 3.9.** Let \( \langle A, \leq \rangle \) be a partially ordered set, and let \( f : A \to A \) be a function. A **fixed-point** of \( f \) is an element \( a \in A \) such that \( f(a) = a \). The **least fixed-point** of \( f \) is an element \( a \in A \) such that \( f(a) = a \), and for every \( b \in A \) such that \( f(b) = b \), then \( a \leq b \).

The following lemma gives sufficient conditions for the existence of least fixed-points. It is one of the key lemmas in denotational semantics.

**Lemma 3.7.** Let \( \langle A, \leq \rangle \) be an \( \omega \)-chain complete poset with least element \( \bot \). Every \( \omega \)-continuous function \( f : A \to A \) has a unique least fixed-point \( x_0 \) given by

\[
x_0 = \bigsqcup f^n(\bot).
\]

Furthermore, for any \( b \in A \) such that \( f(b) \leq b \), then \( x_0 \leq b \).

**Proof.** First, we prove that the sequence

\[
\bot, f(\bot), f^2(\bot), \ldots, f^n(\bot), \ldots
\]

is an \( \omega \)-chain. This is shown by induction on \( n \). Since \( \bot \) is the least element of \( A \), we have \( \bot \leq f(\bot) \). Assuming by induction that \( f^n(\bot) \leq f^{n+1}(\bot) \), since \( f \) is \( \omega \)-continuous, it is monotonic, and thus we get \( f^{n+1}(\bot) \leq f^{n+2}(\bot) \), as desired.

Since \( A \) is an \( \omega \)-chain complete poset, the \( \omega \)-chain \( (f^n(\bot)) \) has a least upper bound

\[
x_0 = \bigsqcup f^n(\bot).
\]

Since \( f \) is \( \omega \)-continuous, we have

\[
f(x_0) = f\left( \bigsqcup f^n(\bot) \right) = \bigsqcup f(f^n(\bot)) = \bigsqcup f^{n+1}(\bot) = x_0,
\]

and \( x_0 \) is indeed a fixed-point of \( f \).

Clearly, if \( f(b) \leq b \) implies that \( x_0 \leq b \), then \( f(b) = b \) implies that \( x_0 \leq b \). Thus, assume that \( f(b) \leq b \) for some \( b \in A \). We prove by induction of \( n \) that \( f^n(\bot) \leq b \). Indeed, \( \bot \leq b \), since \( \bot \) is the least element of \( A \). Assuming by induction that \( f^n(\bot) \leq b \), by monotonicity of \( f \), we get

\[
f(f^n(\bot)) \leq f(b),
\]

and since \( f(b) \leq b \), this yields

\[
f^{n+1}(\bot) \leq b.
\]

Since \( f^n(\bot) \leq b \) for all \( n \geq 0 \), we have

\[
x_0 = \bigsqcup f^n(\bot) \leq b.
\]

\( \square \)
The second part of lemma 3.7 is very useful to prove that functions have the same least fixed-point. For example, under the conditions of lemma 3.7, if \( g: A \to A \) is another \( \omega \)-chain continuous function, letting \( x_0 \) be the least fixed-point of \( f \) and \( y_0 \) be the least fixed-point of \( g \), if \( f(y_0) \leq y_0 \) and \( g(x_0) \leq x_0 \), we can deduce that \( x_0 = y_0 \). Indeed, since \( f(y_0) \leq y_0 \) and \( x_0 \) is the least fixed-point of \( f \), we get \( x_0 \leq y_0 \), and since \( g(x_0) \leq x_0 \) and \( y_0 \) is the least fixed-point of \( g \), we get \( y_0 \leq x_0 \), and therefore \( x_0 = y_0 \).

Lemma 3.7 also shows that the least fixed-point \( x_0 \) of \( f \) can be approximated as much as desired, using the sequence \( (f^n(\bot)) \). We will now apply this fact to context-free grammars.

For this, we need to show how a context-free grammar \( G = (V, \Sigma, P, S) \) with \( m \) nonterminals induces an \( \omega \)-continuous map \( \Phi_G: 2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*} \to 2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*} \).

### 3.8 Context-Free Languages as Least Fixed-Points

Given a context-free grammar \( G = (V, \Sigma, P, S) \) with \( m \) nonterminals \( A_1, \ldots, A_m \), grouping all the productions having the same left-hand side, the grammar \( G \) can be concisely written as

\[
\begin{align*}
A_1 \to & \alpha_{1,1} + \cdots + \alpha_{1,n_1}, \\
\ldots \to & \cdots \\
A_i \to & \alpha_{i,1} + \cdots + \alpha_{i,n_i}, \\
\ldots \to & \cdots \\
A_m \to & \alpha_{m,1} + \cdots + \alpha_{m,n_m}.
\end{align*}
\]

Given any set \( A \), let \( P_{\text{fin}}(A) \) be the set of finite subsets of \( A \).

**Definition 3.10.** Let \( G = (V, \Sigma, P, S) \) be a context-free grammar with \( m \) nonterminals \( A_1, \ldots, A_m \). For any \( m \)-tuple \( \Lambda = (L_1, \ldots, L_m) \) of languages \( L_i \subseteq \Sigma^* \), we define the function

\[
\Phi[\Lambda]: P_{\text{fin}}(V^*) \to 2^{\Sigma^*}
\]

inductively as follows:

\[
\begin{align*}
\Phi[\Lambda](\emptyset) & = \emptyset, \\
\Phi[\Lambda](\{\epsilon\}) & = \{\epsilon\}, \\
\Phi[\Lambda](\{a\}) & = \{a\}, \quad \text{if } a \in \Sigma, \\
\Phi[\Lambda](\{A_i\}) & = L_i, \quad \text{if } A_i \in N, \\
\Phi[\Lambda](\{\alpha X\}) & = \Phi[\Lambda](\{\alpha\})\Phi[\Lambda](\{X\}), \quad \text{if } \alpha \in V^+, X \in V, \\
\Phi[\Lambda](Q \cup \{\alpha\}) & = \Phi[\Lambda](Q) \cup \Phi[\Lambda](\{\alpha\}), \quad \text{if } Q \in P_{\text{fin}}(V^*), Q \neq \emptyset, \alpha \in V^*, \alpha \notin Q.
\end{align*}
\]
Then, writing the grammar $G$ as
\[
A_1 \rightarrow \alpha_{1,1} + \cdots + \alpha_{1,n_1}, \\
\vdots \rightarrow \vdots \\
A_i \rightarrow \alpha_{i,1} + \cdots + \alpha_{i,n_i}, \\
\vdots \rightarrow \vdots \\
A_m \rightarrow \alpha_{m,1} + \cdots + \alpha_{m,n_m},
\]
we define the map
\[
\Phi_G : 2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*} \rightarrow 2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}
\]
such that
\[
\Phi_G(L_1, \ldots, L_m) = (\Phi[\Lambda](\{\alpha_{1,1}, \ldots, \alpha_{1,n_1}\}), \ldots, \Phi[\Lambda](\{\alpha_{m,1}, \ldots, \alpha_{m,n_m}\}))
\]
for all $\Lambda = (L_1, \ldots, L_m) \in 2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}$.

One should verify that the map $\Phi[\Lambda]$ is well defined, but this is easy. The following lemma is easily shown:

**Lemma 3.8.** Given a context-free grammar $G = (V, \Sigma, P, S)$ with $m$ nonterminals $A_1, \ldots, A_m$, the map
\[
\Phi_G : 2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*} \rightarrow 2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}
\]
is $\omega$-continuous.

Now, $2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}$ is an $\omega$-chain complete poset, and the map $\Phi_G$ is $\omega$-continuous. Thus, by lemma 3.7, the map $\Phi_G$ has a least-fixed point. It turns out that the components of this least fixed-point are precisely the languages generated by the grammars $(V, \Sigma, P, A_i)$. Before proving this fact, let us give an example illustrating it.

**Example.** Consider the grammar $G = (\{A, B, a, b\}, \{a, b\}, P, A)$ defined by the rules
\[
A \rightarrow BB + ab, \\
B \rightarrow aBb + ab.
\]

The least fixed-point of $\Phi_G$ is the least upper bound of the chain
\[
(\Phi^n_G(\emptyset, \emptyset)) = ((\Phi^n_{G,A}(\emptyset, \emptyset), \Phi^n_{G,B}(\emptyset, \emptyset)),
\]
where
\[
\Phi^0_{G,A}(\emptyset, \emptyset) = \Phi^0_{G,B}(\emptyset, \emptyset) = \emptyset,
\]
and

\[ \Phi_{G,A}^{n+1}(\emptyset, \emptyset) = \Phi_{G,B}^n(\emptyset, \emptyset) \Phi_{G,B}(\emptyset, \emptyset) \cup \{ab\}, \]

\[ \Phi_{G,B}^{n+1}(\emptyset, \emptyset) = a \Phi_{G,B}^n(\emptyset, \emptyset) b \cup \{ab\}. \]

It is easy to verify that

\[ \Phi_1^{G,A}(\emptyset, \emptyset) = \{ab\}, \]

\[ \Phi_1^{G,B}(\emptyset, \emptyset) = \{ab\}, \]

\[ \Phi_2^{G,A}(\emptyset, \emptyset) = \{ab, abab\}, \]

\[ \Phi_2^{G,B}(\emptyset, \emptyset) = \{ab, aabb\}, \]

\[ \Phi_3^{G,A}(\emptyset, \emptyset) = \{ab, abab, abaabb, aabbab, aabbaabb\}, \]

\[ \Phi_3^{G,B}(\emptyset, \emptyset) = \{ab, aabb, aaabbb\}. \]

By induction, we can easily prove that the two components of the least fixed-point are the languages

\[ L_A = \{a^m b^n a^n b^n \mid m, n \geq 1\} \cup \{ab\} \quad \text{and} \quad L_B = \{a^n b^n \mid n \geq 1\}. \]

Letting \( G_A = (\{A, B, a, b\}, \{a, b\}, P, A) \) and \( G_B = (\{A, B, a, b\}, \{a, b\}, P, B) \), it is indeed true that \( L_A = L(G_A) \) and \( L_B = L(G_B) \).

We have the following theorem due to Ginsburg and Rice:

**Theorem 3.9.** Given a context-free grammar \( G = (V, \Sigma, P, S) \) with \( m \) nonterminals \( A_1, \ldots, A_m \), the least fixed-point of the map \( \Phi_G \) is the \( m \)-tuple of languages

\[ (L(G_{A_1}), \ldots, L(G_{A_m})), \]

where \( G_{A_i} = (V, \Sigma, P, A_i) \).

**Proof.** Writing \( G \) as

\[ A_1 \rightarrow \alpha_{1,1} + \cdots + \alpha_{1,n_1}, \]

\[
\ldots \rightarrow \ldots
\]

\[ A_i \rightarrow \alpha_{i,1} + \cdots + \alpha_{i,n_i}, \]

\[
\ldots \rightarrow \ldots
\]

\[ A_m \rightarrow \alpha_{m,1} + \cdots + \alpha_{m,n_m}, \]

let \( M = \max\{|\alpha_{i,j}|\} \) be the maximum length of right-hand sides of rules in \( P \). Let

\[ \Phi_G^n(\emptyset, \ldots, \emptyset) = (\Phi_{G,1}^n(\emptyset, \ldots, \emptyset), \ldots, \Phi_{G,m}^n(\emptyset, \ldots, \emptyset)). \]
Then, for any \( w \in \Sigma^* \), observe that
\[
w \in \Phi_{G,i}^1(\emptyset, \ldots, \emptyset)\]
iff there is some rule \( A_i \rightarrow \alpha_{i,j} \) with \( w = \alpha_{i,j} \), and that
\[
w \in \Phi_{G,i}^n(\emptyset, \ldots, \emptyset)\]
for some \( n \geq 2 \) iff there is some rule \( A_i \rightarrow \alpha_{i,j} \) with \( \alpha_{i,j} \) of the form
\[
\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},
\]
where \( u_1, \ldots, u_{k+1} \in \Sigma^* \), \( k \geq 1 \), and some \( w_1, \ldots, w_k \in \Sigma^* \) such that
\[
w_h \in \Phi_{G,j_h}^{n-1}(\emptyset, \ldots, \emptyset),
\]
and
\[
w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}.
\]

We prove the following two claims.

**Claim 1:** For every \( w \in \Sigma^* \), if \( A_i \xrightarrow{n} w \), then \( w \in \Phi_{G,i}^p(\emptyset, \ldots, \emptyset) \), for some \( p \geq 1 \).

**Claim 2:** For every \( w \in \Sigma^* \), if \( w \in \Phi_{G,i}^n(\emptyset, \ldots, \emptyset) \), with \( n \geq 1 \), then \( A_i \xrightarrow{p} w \) for some \( p \leq (M + 1)^{n-1} \).

**Proof of Claim 1.** We proceed by induction on \( n \). If \( A_i \xrightarrow{1} w \), then \( w = \alpha_{i,j} \) for some rule \( A \rightarrow \alpha_{i,j} \), and by the remark just before the claim, \( w \in \Phi_{G,i}^1(\emptyset, \ldots, \emptyset) \).

If \( A_i \xrightarrow{n+1} w \) with \( n \geq 1 \), then
\[
A_i \xrightarrow{n} \alpha_{i,j} \xrightarrow{} w
\]
for some rule \( A_i \rightarrow \alpha_{i,j} \). If
\[
\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},
\]
where \( u_1, \ldots, u_{k+1} \in \Sigma^* \), \( k \geq 1 \), then \( A_{j_h} \xrightarrow{n_h} w_h \), where \( n_h \leq n \), and
\[
w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}
\]
for some \( w_1, \ldots, w_k \in \Sigma^* \). By the induction hypothesis,
\[
w_h \in \Phi_{G,j_h}^{p_h}(\emptyset, \ldots, \emptyset),
\]
for some \( p_h \geq 1 \), for every \( h, 1 \leq h \leq k \). Letting \( p = \max\{p_1, \ldots, p_k\} \), since each sequence \( (\Phi_{G,i}^j(\emptyset, \ldots, \emptyset)) \) is an \( \omega \)-chain, we have \( w_h \in \Phi_{G,j_h}^p(\emptyset, \ldots, \emptyset) \) for every \( h, 1 \leq h \leq k \), and by the remark just before the claim, \( w \in \Phi_{G,i}^{p+1}(\emptyset, \ldots, \emptyset) \). \( \square \)
Proof of Claim 2. We proceed by induction on $n$. If $w \in \Phi_{G,i}^1(\emptyset, \ldots, \emptyset)$, by the remark just before the claim, then $w = \alpha_{i,j}$ for some rule $A \rightarrow \alpha_{i,j}$, and $A_i \xRightarrow{1} w$.

If $w \in \Phi_{G,i}^n(\emptyset, \ldots, \emptyset)$ for some $n \geq 2$, then there is some rule $A_i \rightarrow \alpha_{i,j}$ with $\alpha_{i,j}$ of the form

$$\alpha_{i,j} = u_1 A_{j_1} u_2 \cdots u_k A_{j_k} u_{k+1},$$

where $u_1, \ldots, u_{k+1} \in \Sigma^*$, $k \geq 1$, and some $w_1, \ldots, w_k \in \Sigma^*$ such that

$$w_h \in \Phi_{G,j}^{n-1}(\emptyset, \ldots, \emptyset),$$

and

$$w = u_1 w_1 u_2 \cdots u_k w_k u_{k+1}.$$

By the induction hypothesis, $A_{j_h} \xRightarrow{p_h} w_h$ with $p_h \leq (M+1)^{n-2}$, and thus

$$A_i \xRightarrow{p} A_{j_1} \xRightarrow{p_{j_1}} u_2 \xRightarrow{p_{j_2}} \cdots \xRightarrow{p_k} u_{k+1} \xRightarrow{p_{j_k}} w,$$

so that $A_i \xRightarrow{p} w$ with

$$p \leq p_1 + \cdots + p_k + 1 \leq M(M+1)^{n-2} + 1 \leq (M+1)^{n-1},$$

since $k \leq M$.  

Combining Claim 1 and Claim 2, we have

$$L(G_{A_i}) = \bigcup_n \Phi_{G,i}^n(\emptyset, \ldots, \emptyset),$$

which proves that the least fixed-point of the map $\Phi_G$ is the $m$-tuple of languages

$$(L(G_{A_1}), \ldots, L(G_{A_m})).$$

We now show how theorem 3.9 can be used to give a short proof that every context-free grammar can be converted to Greibach Normal Form.

### 3.9 Least Fixed-Points and the Greibach Normal Form

The hard part in converting a grammar $G = (V, \Sigma, P, S)$ to Greibach Normal Form is to convert it to a grammar in so-called weak Greibach Normal Form, where the productions are of the form

$$A \rightarrow a\alpha, \quad \text{or}$$

$$S \rightarrow \epsilon,$$
where \( a \in \Sigma \), \( \alpha \in V^* \), and if \( S \to \epsilon \) is a rule, then \( S \) does not occur on the right-hand side of any rule. Indeed, if we first convert \( G \) to Chomsky Normal Form, it turns out that we will get rules of the form \( A \to aBC \), \( A \to aB \) or \( A \to a \).

Using the algorithm for eliminating \( \epsilon \)-rules and chain rules, we can first convert the original grammar to a grammar with no chain rules and no \( \epsilon \)-rules except possibly \( S \to \epsilon \), in which case, \( S \) does not appear on the right-hand side of rules. Thus, for the purpose of converting to weak Greibach Normal Form, we can assume that we are dealing with grammars without chain rules and without \( \epsilon \)-rules. Let us also assume that we computed the set \( T(G) \) of nonterminals that actually derive some terminal string, and that useless productions involving symbols not in \( T(G) \) have been deleted.

Let us explain the idea of the conversion using the following grammar:

\[
A \to AaB + BB + b. \\
B \to Bd + BAAa + aA + c.
\]

The first step is to group the right-hand sides \( \alpha \) into two categories: those whose leftmost symbol is a terminal (\( \alpha \in \Sigma V^* \)) and those whose leftmost symbol is a nonterminal (\( \alpha \in NV^* \)). It is also convenient to adopt a matrix notation, and we can write the above grammar as

\[
(A, B) = (A, B) \begin{pmatrix}
aB & \emptyset \\
B & \{d, Aa\}
\end{pmatrix} + (b, \{aA, c\})
\]

Thus, we are dealing with matrices (and row vectors) whose entries are finite subsets of \( V^* \). For notational simplicity, braces around singleton sets are omitted. The finite subsets of \( V^* \) form a semiring, where addition is union, and multiplication is concatenation. Addition and multiplication of matrices are as usual, except that the semiring operations are used. We will also consider matrices whose entries are languages over \( \Sigma \). Again, the languages over \( \Sigma \) form a semiring, where addition is union, and multiplication is concatenation. The identity element for addition is \( \emptyset \), and the identity element for multiplication is \( \{\epsilon\} \). As above, addition and multiplication of matrices are as usual, except that the semiring operations are used. For example, given any languages \( A_{i,j} \) and \( B_{i,j} \) over \( \Sigma \), where \( i, j \in \{1, 2\} \), we have

\[
\begin{pmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{pmatrix} \begin{pmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{pmatrix} = \begin{pmatrix}
A_{1,1}B_{1,1} \cup A_{1,2}B_{2,1} & A_{1,1}B_{1,2} \cup A_{1,2}B_{2,2} \\
A_{2,1}B_{1,1} \cup A_{2,2}B_{2,1} & A_{2,1}B_{1,2} \cup A_{2,2}B_{2,2}
\end{pmatrix}
\]

Letting \( X = (A, B), K = (b, \{aA, c\}) \), and

\[
H = \begin{pmatrix}
aB & \emptyset \\
B & \{d, Aa\}
\end{pmatrix}
\]
the above grammar can be concisely written as

\[ X = XH + K. \]

More generally, given any context-free grammar \( G = (V, \Sigma, P, S) \) with \( m \) nonterminals \( A_1, \ldots, A_m \), assuming that there are no chain rules, no \( \epsilon \)-rules, and that every nonterminal belongs to \( T(G) \), letting

\[ X = (A_1, \ldots, A_m), \]

we can write \( G \) as

\[ X = XH + K, \]

for some appropriate \( m \times m \) matrix \( H \) in which every entry contains a set (possibly empty) of strings in \( V^+ \), and some row vector \( K \) in which every entry contains a set (possibly empty) of strings \( \alpha \) each beginning with a terminal \((\alpha \in \Sigma V^*)\).

Given an \( m \times m \) square matrix \( A = (A_{i,j}) \) of languages over \( \Sigma \), we can define the matrix \( A^* \) whose entry \( A^*_{i,j} \) is given by

\[ A^*_{i,j} = \bigcup_{n \geq 0} A^n_{i,j}, \]

where \( A^0 = Id_m \), the identity matrix, and \( A^n \) is the \( n \)-th power of \( A \). Similarly, we define \( A^+ \) where

\[ A^+_{i,j} = \bigcup_{n \geq 1} A^n_{i,j}. \]

Given a matrix \( A \) where the entries are finite subset of \( V^* \), where \( N = \{A_1, \ldots, A_m\} \), for any \( m \)-tuple \( \Lambda = (L_1, \ldots, L_m) \) of languages over \( \Sigma \), we let

\[ \Phi[\Lambda](A) = (\Phi[\Lambda](A_{i,j})). \]

Given a system \( X = XH + K \) where \( H \) is an \( m \times m \) matrix and \( X, K \) are row matrices, if \( H \) and \( K \) do not contain any nonterminals, we claim that the least fixed-point of the grammar \( G \) associated with \( X = XH + K \) is \( KH^* \). This is easily seen by computing the approximations \( X^n = \Phi[\Lambda]^n(\emptyset, \ldots, \emptyset) \). Indeed, \( X^0 = K \), and

\[ X^n = KH^n + KH^{n-1} + \cdots + KH + K = K(H^n + H^{n-1} + \cdots + H + I_m). \]

Similarly, if \( Y \) is an \( m \times m \) matrix of nonterminals, the least fixed-point of the grammar associated with \( Y = HY + H \) is \( H^+ \) (provided that \( H \) does not contain any nonterminals).

Given any context-free grammar \( G = (V, \Sigma, P, S) \) with \( m \) nonterminals \( A_1, \ldots, A_m \), writing \( G \) as \( X = XH + K \) as explained earlier, we can form another grammar \( GH \) by creating \( m^2 \) new nonterminals \( Y_{i,j} \), where the rules of this new grammar are defined by the system of two matrix equations

\[ X = KY + K, \]
\[ Y = HY + H, \]
where $Y = (Y_{i,j})$.

The following lemma is the key to the Greibach Normal Form.

Lemma 3.10. Given any context-free grammar $G = (V, \Sigma, P, S)$ with $m$ nonterminals $A_1, \ldots, A_m$, writing $G$ as

$$X = XH + K$$

as explained earlier, if $GH$ is the grammar defined by the system of two matrix equations

$$X = KY + K,$$
$$Y = HY + H,$$

as explained above, then the components in $X$ of the least-fixed points of the maps $\Phi_G$ and $\Phi_{GH}$ are equal.

Proof. Let $U$ be the least-fixed point of $\Phi_G$, and let $(V, W)$ be the least fixed-point of $\Phi_{GH}$. We shall prove that $U = V$. For notational simplicity, let us denote $\Phi[U](H)$ as $H[U]$ and $\Phi[U](K)$ as $K[U]$.

Since $U$ is the least fixed-point of $X = XH + K$, we have

$$U = UH[U] + K[U].$$

Since $H[U]$ and $K[U]$ do not contain any nonterminals, by a previous remark, $K[U]H^*[U]$ is the least-fixed point of $X = XH[U] + K[U]$, and thus,

$$K[U]H^*[U] \leq U.$$

On the other hand, by monotonicity,


and since $U$ is the least fixed-point of $X = XH + K$,

$$U \leq K[U]H^*[U].$$

Therefore, $U = K[U]H^*[U]$. We can prove in a similar manner that $W = H[V]^+$.

Let $Z = H[U]^+$. We have


and


and since $(V, W)$ is the least fixed-point of $X = KY + K$ and $Y = HY + H$, we get $V \leq U$ and $W \leq H[U]^+$. 


We also have
and
and since \( U \) is the least fixed-point of \( X = XH + K \), we get \( U \leq V \). Therefore, \( U = V \), as claimed.

Note that the above lemma actually applies to any grammar. Applying lemma 3.10 to our example grammar, we get the following new grammar:

\[
(A, B) = (b, \{aA, c\}) (Y_1 Y_2) + (b, \{aA, c\}),

\[
\begin{pmatrix}
Y_1 & Y_2 \\
Y_3 & Y_4
\end{pmatrix}
= \begin{pmatrix}
aB & \emptyset \\
B & \{d, Aa\}
\end{pmatrix}
\begin{pmatrix}
Y_1 & Y_2 \\
Y_3 & Y_4
\end{pmatrix}
+ \begin{pmatrix}
aB & \emptyset \\
B & \{d, Aa\}
\end{pmatrix}
\]

There are still some nonterminals appearing as leftmost symbols, but using the equations defining \( A \) and \( B \), we can replace \( A \) with
\[ \{bY_1, aAY_3, cY_3, b\} \]
and \( B \) with
\[ \{bY_2, aAY_4, cY_4, aA, c\}, \]
obtaining a system in weak Greibach Normal Form. This amounts to converting the matrix
\[ H = \begin{pmatrix}
aB & \emptyset \\
B & \{d, Aa\}
\end{pmatrix} \]
to the matrix
\[ L = \begin{pmatrix}
aB & \emptyset \\
\{bY_2, aAY_4, cY_4, aA, c\} & \{d, bY_1a, aAY_3a, cY_3a, ba\}
\end{pmatrix} \]

The weak Greibach Normal Form corresponds to the new system
\[ X = KY + K, \]
\[ Y = LY + L. \]

This method works in general for any input grammar with no \( \epsilon \)-rules, no chain rules, and such that every nonterminal belongs to \( T(G) \). Under these conditions, the row vector \( K \)
contains some nonempty entry, all strings in $K$ are in $\Sigma V^*$, and all strings in $H$ are in $V^+$. After obtaining the grammar $GH$ defined by the system
\[
X = KY + K, \\
Y = HY + H,
\]
we use the system $X = KY + K$ to express every nonterminal $A_i$ in terms of expressions containing strings $\alpha_{i,j}$ involving a terminal as the leftmost symbol ($\alpha_{i,j} \in \Sigma V^*$), and we replace all leftmost occurrences of nonterminals in $H$ (occurrences $A_i$ in strings of the form $A_i \beta$, where $\beta \in V^*$) using the above expressions. In this fashion, we obtain a matrix $L$, and it is immediately shown that the system
\[
X = KY + K, \\
Y = LY + L,
\]
generates the same tuple of languages. Furthermore, this last system corresponds to a weak Greibach Normal Form.

It we start with a grammar in Chomsky Normal Form (with no production $S \rightarrow \epsilon$) such that every nonterminal belongs to $T(G)$, we actually get a Greibach Normal Form (the entries in $K$ are terminals, and the entries in $H$ are nonterminals). Thus, we have justified lemma 3.6. The method is also quite economical, since it introduces only $m^2$ new nonterminals. However, the resulting grammar may contain some useless nonterminals.

### 3.10 Tree Domains and Gorn Trees

Derivation trees play a very important role in parsing theory and in the proof of a strong version of the pumping lemma for the context-free languages known as Ogden’s lemma. Thus, it is important to define derivation trees rigorously. We do so using Gorn trees.

Let $\mathbb{N}_+ = \{1, 2, 3, \ldots \}$.

**Definition 3.11.** A tree domain $D$ is a nonempty subset of strings in $\mathbb{N}_+^*$ satisfying the conditions:

1. For all $u, v \in \mathbb{N}_+^*$, if $uv \in D$, then $u \in D$.
2. For all $u \in \mathbb{N}_+^*$, for every $i \in \mathbb{N}_+$, if $ui \in D$ then $uj \in D$ for every $j$, $1 \leq j \leq i$.

The tree domain
\[
D = \{\epsilon, 1, 2, 11, 21, 22, 221, 222, 2211\}
\]
is represented as follows:
A tree labeled with symbols from a set $\Delta$ is defined as follows.

**Definition 3.12.** Given a set $\Delta$ of labels, a $\Delta$-tree (for short, a *tree*) is a total function $t : D \rightarrow \Delta$, where $D$ is a tree domain.

The domain of a tree $t$ is denoted as $\text{dom}(t)$. Every string $u \in \text{dom}(t)$ is called a *tree address* or a *node*.

Let $\Delta = \{f, g, h, a, b\}$. The tree $t : D \rightarrow \Delta$, where $D$ is the tree domain of the previous example and $t$ is the function whose graph is

$$\{(\epsilon, f), (1, h), (2, g), (11, a), (21, a), (22, f), (221, h), (222, b), (2211, a)\}$$

is represented as follows:

![Diagram]

The *outdegree* (sometimes called *ramification*) $r(u)$ of a node $u$ is the cardinality of the set

$$\{i \mid ui \in \text{dom}(t)\}.$$ 

Note that the outdegree of a node can be infinite. Most of the trees that we shall consider will be *finite-branching*, that is, for every node $u$, $r(u)$ will be an integer, and hence finite. If the outdegree of all nodes in a tree is bounded by $n$, then we can view the domain of the tree as being defined over $\{1, 2, \ldots, n\}^*$. 
A node of outdegree 0 is called a \textit{leaf}. The node whose address is \(\epsilon\) is called the \textit{root} of the tree. A tree is \textit{finite} if its domain \(\text{dom}(t)\) is finite. Given a node \(u\) in \(\text{dom}(t)\), every node of the form \(ui\) in \(\text{dom}(t)\) with \(i \in \mathbb{N}_+\) is called a \textit{son} (or \textit{immediate successor}) of \(u\).

Tree addresses are totally ordered \textit{lexicographically}: \(u \leq v\) if either \(u\) is a prefix of \(v\) or, there exist strings \(x, y, z \in \mathbb{N}_+^*\) and \(i, j \in \mathbb{N}_+\), with \(i < j\), such that \(u = xiy\) and \(v = xjz\).

In the first case, we say that \(u\) is an \textit{ancestor} (or \textit{predecessor}) of \(v\) (or \(u\) \textit{dominates} \(v\)) and in the second case, that \(u\) is \textit{to the left} of \(v\).

If \(y = \epsilon\) and \(z = \epsilon\), we say that \(xi\) is a \textit{left brother} (or \textit{left sibling}) of \(xj\), \((i < j)\). Two tree addresses \(u\) and \(v\) are \textit{independent} if \(u\) is not a prefix of \(v\) and \(v\) is not a prefix of \(u\).

Given a finite tree \(t\), the \textit{yield} of \(t\) is the string
\[ t(u_1)t(u_2)\cdots t(u_k), \]
where \(u_1, u_2, \ldots, u_k\) is the sequence of leaves of \(t\) in lexicographic order.

For example, the yield of the tree below is \(\textit{aaab}\):

\[
\begin{array}{c}
  f \\
  \downarrow \quad \downarrow \\
  h \\
  \downarrow \quad \downarrow \\
  a \quad a \\
  \downarrow \quad \downarrow \\
  f \\
  \downarrow \\
  b \\
  \downarrow \\
  a
\end{array}
\]

Given a finite tree \(t\), the \textit{depth} of \(t\) is the integer
\[ d(t) = \max\{|u| \mid u \in \text{dom}(t)|. \]

Given a tree \(t\) and a node \(u\) in \(\text{dom}(t)\), the \textit{subtree rooted at} \(u\) is the tree \(t/u\), whose domain is the set
\[ \{v \mid uv \in \text{dom}(t)\} \]
and such that \(t/u(v) = t(uv)\) for all \(v\) in \(\text{dom}(t/u)\).

Another important operation is the operation of tree replacement (or tree substitution).

\textbf{Definition 3.13.} Given two trees \(t_1\) and \(t_2\) and a tree address \(u\) in \(t_1\), the \textit{result of substituting} \(t_2\) at \(u\) in \(t_1\), denoted by \(t_1[u \leftarrow t_2]\), is the function whose graph is the set of pairs
\[ \{(v, t_1(v)) \mid v \in \text{dom}(t_1), u \text{ is not a prefix of } v\} \cup \{(uv, t_2(v)) \mid v \in \text{dom}(t_2)\}. \]
Let $t_1$ and $t_2$ be the trees defined by the following diagrams:

**Tree $t_1$**

```
  f
 /\   /
g /  /
\h\ / h
 a   a
```

**Tree $t_2$**

```
 g
 /\   /
\ a\  b
```

The tree $t_1[22 \leftarrow t_2]$ is defined by the following diagram:

```
  f
 /\   /
g /  /
\h\ / h
 a   a
```

We can now define derivation trees and relate derivations to derivation trees.

### 3.11 Derivations Trees

**Definition 3.14.** Given a context-free grammar $G = (V, \Sigma, P, S)$, for any $A \in N$, an $A$-derivation tree for $G$ is a $(V \cup \{\epsilon\})$-tree $t$ (a tree with set of labels $(V \cup \{\epsilon\})$) such that:

1. $t(\epsilon) = A$;

2. For every nonleaf node $u \in \text{dom}(t)$, if $u_1, \ldots, u_k$ are the successors of $u$, then either there is a production $B \rightarrow X_1 \cdots X_k$ in $P$ such that $t(u) = B$ and $t(u_i) = X_i$ for all $i$, $1 \leq i \leq k$, or $B \rightarrow \epsilon \in P$, $t(u) = B$ and $t(u1) = \epsilon$. A complete derivation (or parse tree) is an $S$-tree whose yield belongs to $\Sigma^*$. 

A derivation tree for the grammar

\[ G_3 = (\{E, T, F, +, *, (, ), a\}, \{+ , *, ( , ) , a\}, P, E), \]

where \( P \) is the set of rules

\[
\begin{align*}
E & \rightarrow E + T, \\
E & \rightarrow T, \\
T & \rightarrow T * F, \\
T & \rightarrow F, \\
F & \rightarrow (E), \\
F & \rightarrow a,
\end{align*}
\]

is shown in Figure 3.1. The yield of the derivation tree is \( a + a * a \).

Figure 3.1: A complete derivation tree

Derivations trees are associated to derivations inductively as follows.

**Definition 3.15.** Given a context-free grammar \( G = (V, \Sigma, P, S) \), for any \( A \in N \), if \( \pi : A \xrightarrow{n} \alpha \) is a derivation in \( G \), we construct an \( A \)-derivation tree \( t_\pi \) with yield \( \alpha \) as follows.

1. If \( n = 0 \), then \( t_\pi \) is the one-node tree such that \( \text{dom}(t_\pi) = \{\epsilon\} \) and \( t_\pi(\epsilon) = A \).

2. If \( \lambda \rightarrow A \xrightarrow{n-1} B \rho \rightarrow \lambda \gamma \rho = \alpha \), then if \( t_1 \) is the \( A \)-derivation tree with yield \( \lambda B \rho \) associated with the derivation \( \lambda \rightarrow A \xrightarrow{n-1} \lambda B \rho \), and if \( t_2 \) is the tree associated with the production \( B \rightarrow \gamma \) (that is, if \( \gamma = X_1 \cdots X_k \)), then \( \text{dom}(t_2) = \{\epsilon, 1, \ldots, k\} \), \( t_2(\epsilon) = B \), and \( t_2(i) = X_i \) for all \( i, 1 \leq i \leq k \), or if \( \gamma = \epsilon \), then \( \text{dom}(t_2) = \{\epsilon, 1\} \), \( t_2(\epsilon) = B \), and \( t_2(1) = \epsilon \), then

\[
t_\pi = t_1[u \leftarrow t_2],
\]

where \( u \) is the address of the leaf labeled \( B \) in \( t_1 \).
The tree $t_\pi$ is the $A$-derivation tree associated with the derivation $A \overset{n}{\Rightarrow} \alpha$.

Given the grammar

$$G_2 = (\{E, +, *, (,), a\}, \{+,*,(,), a\}, P, E),$$

where $P$ is the set of rules

- $E \rightarrow E + E$,
- $E \rightarrow E \ast E$,
- $E \rightarrow (E)$,
- $E \rightarrow a$,

the parse trees associated with two derivations of the string $a + a \ast a$ are shown in Figure 3.2:

![Parse Trees](image)

Figure 3.2: Two derivation trees for $a + a \ast a$

The following lemma is easily shown.

**Lemma 3.11.** Let $G = (V, \Sigma, P, S)$ be a context-free grammar. For any derivation $A \overset{n}{\Rightarrow} \alpha$, there is a unique $A$-derivation tree associated with this derivation, with yield $\alpha$. Conversely, for any $A$-derivation tree $t$ with yield $\alpha$, there is a unique leftmost derivation $A \overset{\text{lm}}{\Rightarrow} \alpha$ in $G$ having $t$ as its associated derivation tree.

We will now prove a strong version of the pumping lemma for context-free languages due to Bill Ogden (1968).

### 3.12 Ogden’s Lemma

Ogden’s lemma states some combinatorial properties of parse trees that are deep enough. The yield $w$ of such a parse tree can be split into 5 substrings $u, v, x, y, z$ such that

$$w = uvxyz,$$
where \( u, v, x, y, z \) satisfy certain conditions. It turns out that we get a more powerful version of the lemma if we allow ourselves to mark certain occurrences of symbols in \( w \) before invoking the lemma. We can imagine that marked occurrences in a nonempty string \( w \) are occurrences of symbols in \( w \) in boldface, or red, or any given color (but one color only). For example, given \( w = aaababbaa \), we can mark the symbols of even index as follows:

\[
\begin{align*}
& a \quad a \\
& a \quad b \\
& a \quad b \\
& a \quad b \\
& a \\
& a \\
\end{align*}
\]

More rigorously, we can define a marking of a nonnull string \( w : \{1, \ldots, n\} \to \Sigma \) as any function \( m : \{1, \ldots, n\} \to \{0, 1\} \). Then, a letter \( w_i \) in \( w \) is a marked occurrence iff \( m(i) = 1 \), and an unmarked occurrence if \( m(i) = 0 \). The number of marked occurrences in \( w \) is equal to

\[
\sum_{i=1}^{n} m(i).
\]

Ogden’s lemma only yields useful information for grammars \( G \) generating an infinite language. We could make this hypothesis, but it seems more elegant to use the precondition that the lemma only applies to strings \( w \in L(D) \) such that \( w \) contains at least \( K \) marked occurrences, for a constant \( K \) large enough. If \( K \) is large enough, \( L(G) \) will indeed be infinite.

**Lemma 3.12.** For every context-free grammar \( G \), there is some integer \( K > 1 \) such that, for every string \( w \in \Sigma^+ \), for every marking of \( w \), if \( w \in L(G) \) and \( w \) contains at least \( K \) marked occurrences, then there exists some decomposition of \( w \) as \( w = uvxyz \), and some \( A \in N \), such that the following properties hold:

1. There are derivations \( S \overset{+}{\Rightarrow} uAz \), \( A \overset{+}{\Rightarrow} vAy \), and \( A \overset{+}{\Rightarrow} x \), so that
   \[
   uv^nxy^nz \in L(G)
   \]
   for all \( n \geq 0 \) (the pumping property);
2. \( x \) contains some marked occurrence;
3. Either (both \( u \) and \( v \) contain some marked occurrence), or (both \( y \) and \( z \) contain some marked occurrence);
4. \( vxy \) contains less than \( K \) marked occurrences.

**Proof.** Let \( t \) be any parse tree for \( w \). We call a leaf of \( t \) a marked leaf if its label is a marked occurrence in the marked string \( w \). The general idea is to make sure that \( K \) is large enough so that parse trees with yield \( w \) contain enough repeated nonterminals along some path from the root to some marked leaf. Let \( r = |N| \), and let

\[
p = \max\{2, \max\{|\alpha| \mid (A \to \alpha) \in P\}\}.
\]
We claim that $K = p^{2r + 3}$ does the job.

The key concept in the proof is the notion of a $B$-node. Given a parse tree $t$, a $B$-node is a node with at least two immediate successors $u_1, u_2$, such that for $i = 1, 2$, either $u_i$ is a marked leaf, or $u_i$ has some marked leaf as a descendant. We construct a path from the root to some marked leaf, so that for every $B$-node, we pick the leftmost successor with the maximum number of marked leaves as descendants. Formally, define a path $(s_0, \ldots, s_n)$ from the root to some marked leaf, so that:

(i) Every node $s_i$ has some marked leaf as a descendant, and $s_0$ is the root of $t$;

(ii) If $s_j$ is in the path, $s_j$ is not a leaf, and $s_j$ has a single immediate descendant which is either a marked leaf or has marked leaves as its descendants, let $s_{j+1}$ be that unique immediate descendant of $s_i$.

(iii) If $s_j$ is a $B$-node in the path, then let $s_{j+1}$ be the leftmost immediate successors of $s_j$ with the maximum number of marked leaves as descendants (assuming that if $s_{j+1}$ is a marked leaf, then it is its own descendant).

(iv) If $s_j$ is a leaf, then it is a marked leaf and $n = j$.

We will show that the path $(s_0, \ldots, s_n)$ contains at least $2r + 3$ $B$-nodes.

*Claim*: For every $i$, $0 \leq i \leq n$, if the path $(s_i, \ldots, s_n)$ contains $b$ $B$-nodes, then $s_i$ has at most $p^b$ marked leaves as descendants.

*Proof*. We proceed by “backward induction”, i.e., by induction on $n - i$. For $i = n$, there are no $B$-nodes, so that $b = 0$, and there is indeed $p^0 = 1$ marked leaf $s_n$. Assume that the claim holds for the path $(s_{i+1}, \ldots, s_n)$.

If $s_i$ is not a $B$-node, then the number $b$ of $B$-nodes in the path $(s_{i+1}, \ldots, s_n)$ is the same as the number of $B$-nodes in the path $(s_i, \ldots, s_n)$, and $s_{i+1}$ is the only immediate successor of $s_i$ having a marked leaf as descendant. By the induction hypothesis, $s_{i+1}$ has at most $p^b$ marked leaves as descendants, and this is also an upper bound on the number of marked leaves which are descendants of $s_i$.

If $s_i$ is a $B$-node, then if there are $b$ $B$-nodes in the path $(s_{i+1}, \ldots, s_n)$, there are $b + 1$ $B$-nodes in the path $(s_i, \ldots, s_n)$. By the induction hypothesis, $s_{i+1}$ has at most $p^b$ marked leaves as descendants. Since $s_i$ is a $B$-node, $s_{i+1}$ was chosen to be the leftmost immediate successor of $s_i$ having the maximum number of marked leaves as descendants. Thus, since the outdegree of $s_i$ is at most $p$, and each of its immediate successors has at most $p^b$ marked leaves as descendants, the node $s_i$ has at most $p^{b+1} = p^{d+1}$ marked leaves as descendants, as desired. □

Applying the claim to $s_0$, since $w$ has at least $K = p^{2r + 3}$ marked occurrences, we have $p^b \geq p^{2r + 3}$, and since $p \geq 2$, we have $b \geq 2r + 3$, and the path $(s_0, \ldots, s_n)$ contains at least $2r + 3$ $B$-nodes (Note that this would not follow if we had $p = 1$).
Let us now select the lowest $2r + 3$ $B$-nodes in the path, $(s_0, \ldots, s_n)$, and denote them $(b_1, \ldots, b_{2r+3})$. Every $B$-node $b_i$ has at least two immediate successors $u_i < v_i$ such that $u_i$ or $v_i$ is on the path $(s_0, \ldots, s_n)$. If the path goes through $u_i$, we say that $b_i$ is a right $B$-node and if the path goes through $v_i$, we say that $b_i$ is a left $B$-node. Since $2r + 3 = r + 2 + r + 1$, either there are $r + 2$ left $B$-nodes or there are $r + 2$ right $B$-nodes in the path $(b_1, \ldots, b_{2r+3})$. Let us assume that there are $r + 2$ left $B$-nodes, the other case being similar.

Let $(d_1, \ldots, d_{r+2})$ be the lowest $r + 2$ left $B$-nodes in the path. Since there are $r + 1$ $B$-nodes in the sequence $(d_2, \ldots, d_{r+2})$, and there are only $r$ distinct nonterminals, there are two nodes $d_i$ and $d_j$, with $2 \leq i < j \leq r + 2$, such that $t(d_i) = t(d_j) = A$, for some $A \in N$. We can assume that $d_i$ is an ancestor of $d_j$, and thus, $d_j = d_i \alpha$, for some $\alpha \neq \epsilon$.

If we prune out the subtree $t/d_i$ rooted at $d_i$ from $t$, we get an $S$-derivation tree having a yield of the form $uAz$, and we have a derivation of the form $S \xrightarrow{+} uAz$, since there are at least $r + 2$ left $B$-nodes on the path, and we are looking at the lowest $r + 1$ left $B$-nodes. Considering the subtree $t/d_i$, pruning out the subtree $t/d_j$ rooted at $\alpha$ in $t/d_i$, we get an $A$-derivation tree having a yield of the form $vAy$, and we have a derivation of the form $A \xrightarrow{+} vAy$. Finally, the subtree $t/d_j$ is an $A$-derivation tree with yield $x$, and we have a derivation $A \xrightarrow{+} x$. This proves (1) of the lemma.

Since $s_n$ is a marked leaf and a descendant of $d_j$, $x$ contains some marked occurrence, proving (2).

Since $d_i$ is a left $B$-node, some left sibling of the immediate successor of $d_i$ on the path has some distinguished leaf in $u$ as a descendant. Similarly, since $d_i$ is a left $B$-node, some left sibling of the immediate successor of $d_i$ on the path has some distinguished leaf in $v$ as a descendant. This proves (3).

$(d_j, \ldots, b_{2r+3})$ has at most $2r + 1$ $B$-nodes, and by the claim shown earlier, $d_j$ has at most $p^{2r+1}$ marked leaves as descendants. Since $p^{2r+1} < p^{2r+3} = K$, this proves (4).

Observe that condition (2) implies that $x \neq \epsilon$, and condition (3) implies that either $u \neq \epsilon$ and $v \neq \epsilon$, or $y \neq \epsilon$ and $z \neq \epsilon$. Thus, the pumping condition (1) implies that the set $\{uv^nxy^n | n \geq 0\}$ is an infinite subset of $L(G)$, and $L(G)$ is indeed infinite, as we mentioned earlier. Note that $K \geq 3$, and in fact, $K \geq 32$. The “standard pumping lemma” due to Bar-Hillel, Perles, and Shamir, is obtained by letting all occurrences be marked in $w \in L(G)$.

**Lemma 3.13.** For every context-free grammar $G$ (without $\epsilon$-rules), there is some integer $K > 1$ such that, for every string $w \in \Sigma^+$, if $w \in L(G)$ and $|w| \geq K$, then there exists some decomposition of $w$ as $w = uvxyz$, and some $A \in N$, such that the following properties hold:

1. There are derivations $S \xrightarrow{+} uAz$, $A \xrightarrow{+} vAy$, and $A \xrightarrow{+} x$, so that $uv^nxy^n z \in L(G)$

for all $n \geq 0$ (the pumping property);
3.12. OGDEN’S LEMMA

(2) \( x \neq \epsilon \);

(3) Either \( v \neq \epsilon \) or \( y \neq \epsilon \);

(4) \(|vxy| \leq K\).

A stronger version could be stated, and we are just following tradition in stating this standard version of the pumping lemma.

Ogden’s lemma or the pumping lemma can be used to show that certain languages are not context-free. The method is to proceed by contradiction, i.e., to assume (contrary to what we wish to prove) that a language \( L \) is indeed context-free, and derive a contradiction of Ogden’s lemma (or of the pumping lemma). Thus, as in the case of the regular languages, it would be helpful to see what the negation of Ogden’s lemma is, and for this, we first state Ogden’s lemma as a logical formula.

For any nonnull string \( w: \{1, \ldots, n\} \rightarrow \Sigma \), for any marking \( m: \{1, \ldots, n\} \rightarrow \{0, 1\} \) of \( w \), for any substring \( y \) of \( w \), where \( w = xyz \), with \(|x| = h \) and \( k = |y| \), the number of marked occurrences in \( y \), denoted as \( |m(y)| \), is defined as

\[
|m(y)| = \sum_{i=h+1}^{i=h+k} m(i).
\]

We will also use the following abbreviations:

\[
nat = \{0, 1, 2, \ldots\},
\]

\[
nat32 = \{32, 33, \ldots\},
\]

\[
A \equiv w = uvxyz,
\]

\[
B \equiv |m(x)| \geq 1,
\]

\[
C \equiv (|m(u)| \geq 1 \land |m(v)| \geq 1) \lor (|m(y)| \geq 1 \land |m(z)| \geq 1),
\]

\[
D \equiv |m(vxy)| < K,
\]

\[
P \equiv \forall n: nat \ (uv^nxy^nz \in L(D)).
\]

Ogden’s lemma can then be stated as

\[
\forall G: \text{CFG} \ \exists K: \ nat32 \ \forall w: \Sigma^* \ \forall m: \text{marking} \ \left( (w \in L(D) \land |m(w)| \geq K) \supset (\exists u, v, x, y, z: \Sigma^* \ A \land B \land C \land D \land P) \right).
\]

Recalling that

\[
\neg(A \land B \land C \land D \land P) \equiv \neg(A \land B \land C \land D) \lor \neg P \iff (A \land B \land C \land D) \supset \neg P
\]

and

\[
\neg(P \supset Q) \equiv P \land \neg Q,
\]
the negation of Ogden’s lemma can be stated as

\[ \exists G: \text{CFG} \forall K: \text{nat32} \exists w: \Sigma^* \exists m: \text{marking} \]

\[ \left( (w \in L(D) \land |m(w)| \geq K) \land (\forall u, v, x, y, z: \Sigma^*(A \land B \land C \land D) \subset \neg P) \right). \]

Since \( -P \equiv \exists n: \text{nat} (uv^nxy^n z \notin L(D)) \), in order to show that Ogden’s lemma is contradicted, one needs to show that for some context-free grammar \( G \), for every \( K \geq 2 \), there is some string \( w \in L(D) \) and some marking \( m \) of \( w \) with at least \( K \) marked occurrences in \( w \), such that for every possible decomposition \( w = uvxyz \) satisfying the constraints \( A \land B \land C \land D \), there is some \( n \geq 0 \) such that \( uv^nxy^n z \notin L(D) \). When proceeding by contradiction, we have a language \( L \) that we are (wrongly) assuming to be context-free and we can use any CFG grammar \( G \) generating \( L \). The creative part of the argument is to pick the right \( w \in L \) and the right marking of \( w \) (not making any assumption on \( K \)).

As an illustration, we show that the language

\[ L = \{ a^n b^n c^n \mid n \geq 1 \} \]

is not context-free. Since \( L \) is infinite, we will be able to use the pumping lemma.

The proof proceeds by contradiction. If \( L \) was context-free, there would be some context-free grammar \( G \) such that \( L = L(G) \), and some constant \( K > 1 \) as in Ogden’s lemma. Let \( w = a^K b^K c^K \), and choose the \( b \)'s as marked occurrences. Then by Ogden’s lemma, \( x \) contains some marked occurrence, and either both \( u, v \) or both \( y, z \) contain some marked occurrence. Assume that both \( u \) and \( v \) contain some \( b \). We have the following situation:

\[ a \cdots ab \cdots b \cdots b \cdots b \cdots b \cdots b \cdots c \cdots c. \]

If we consider the string \( uvvxyyz \), the number of \( a \)'s is still \( K \), but the number of \( b \)'s is strictly greater than \( K \) since \( v \) contains at least one \( b \), and thus \( uvvxyyz \notin L \), a contradiction.

If both \( y \) and \( z \) contain some \( b \) we will also reach a contradiction because in the string \( uvvxyyz \), the number of \( c \)'s is still \( K \), but the number of \( b \)'s is strictly greater than \( K \). Having reached a contradiction in all cases, we conclude that \( L \) is not context-free.

Let us now show that the language

\[ L = \{ a^m b^n c^m d^n \mid m, n \geq 1 \} \]

is not context-free.

Again, we proceed by contradiction. This time, let

\[ w = a^K b^K c^K d^K, \]
3.12. OGDEN’S LEMMA

where the b’s and c’s are marked occurrences.

By Ogden’s lemma, either both u, v contain some marked occurrence, or both y, z contain some marked occurrence, and x contains some marked occurrence. Let us first consider the case where both u, v contain some marked occurrence.

If v contains some b, since uvvxyyz ∈ L, v must contain only b’s, since otherwise we would have a bad string in L, and we have the following situation:

\[
\begin{array}{c}
\cdots a \cdots b b \cdots b b \cdots b c \cdots c \cdots d \cdots d
\end{array}
\]

Since uvvxyyz ∈ L, the only way to preserve an equal number of b’s and d’s is to have \(y \in d^+\). But then, vxy contains \(c^K\), which contradicts (4) of Ogden’s lemma.

If v contains some c, since x also contains some marked occurrence, it must be some c, and v contains only c’s and we have the following situation:

\[
\begin{array}{c}
\cdots a \cdots b c c \cdots c c \cdots c c \cdots c \cdots d \cdots d
\end{array}
\]

Since uvvxyyz ∈ L and the number of a’s is still K whereas the number of c’s is strictly more than K, this case is impossible.

Let us now consider the case where both y, z contain some marked occurrence. Reasoning as before, the only possibility is that v ∈ a^+ and y ∈ c^+:

\[
\begin{array}{c}
\cdots a a \cdots a a b a \cdots b c c \cdots c c \cdots c c \cdots c \cdots d \cdots d
\end{array}
\]

But then, vxy contains \(b^K\), which contradicts (4) of Ogden’s Lemma. Since a contradiction was obtained in all cases, L is not context-free.

Ogden’s lemma can also be used to show that the context-free language

\[\{a^m b^n c^n \mid m, n \geq 1\} \cup \{a^m b^n c^n \mid m, n \geq 1\}\]

is inherently ambiguous. The proof is quite involved.

Another corollary of the pumping lemma is that it is decidable whether a context-free grammar generates an infinite language.

**Lemma 3.14.** Given any context-free grammar, G, if K is the constant of Ogden’s lemma, then the following equivalence holds:

L(G) is infinite iff there is some w ∈ L(G) such that K ≤ |w| < 2K.
Proof. Let $K = p^{2r+3}$ be the constant from the proof of Lemma 3.12. If there is some $w \in L(G)$ such that $|w| \geq K$, we already observed that the pumping lemma implies that $L(G)$ contains an infinite subset of the form \{uv^nxy^nz \mid n \geq 0\}. Conversely, assume that $L(G)$ is infinite. If $|w| < K$ for all $w \in L(G)$, then $L(G)$ is finite. Thus, there is some $w \in L(G)$ such that $|w| \geq K$. Let $w \in L(G)$ be a minimal string such that $|w| \geq K$. By the pumping lemma, we can write $w$ as $w = uvxyxz$, where $x \neq \epsilon$, $vy \neq \epsilon$, and $|vxy| \leq K$. By the pumping property, $uxz \in L(G)$. If $|w| \geq 2K$, then

$$|uxz| = |uvxyz| - |vy| > |uvxy| - |vxy| \geq 2K - K = K,$$

and $|uxz| < |uvxyz|$, contradicting the minimality of $w$. Thus, we must have $|w| < 2K$. \qed

In particular, if $G$ is in Chomsky Normal Form, it can be shown that we just have to consider derivations of length at most $4K - 3$. 
Chapter 4

A Survey of $LR$-Parsing Methods

In this chapter, we give a brief survey on $LR$-parsing methods. We begin with the definition of characteristic strings and the construction of Knuth’s $LR(0)$-characteristic automaton. Next, we describe the shift/reduce algorithm. The need for lookahead sets is motivated by the resolution of conflicts. A unified method for computing FIRST, FOLLOW and LALR(1) lookahead sets is presented. The method uses a same graph algorithm Traverse which finds all nodes reachable from a given node and computes the union of predefined sets assigned to these nodes. Hence, the only difference between the various algorithms for computing FIRST, FOLLOW and LALR(1) lookahead sets lies in the fact that the initial sets and the graphs are computed in different ways. The method can be viewed as an efficient way for solving a set of simultaneously recursive equations with set variables. The method is inspired by DeRemer and Pennello’s method for computing LALR(1) lookahead sets. However, DeRemer and Pennello use a more sophisticated graph algorithm for finding strongly connected components. We use a slightly less efficient but simpler algorithm (a depth-first search). We conclude with a brief presentation of $LR(1)$ parsers.

4.1 $LR(0)$-Characteristic Automata

The purpose of $LR$-parsing, invented by D. Knuth in the mid sixties, is the following: Given a context-free grammar $G$, for any terminal string $w \in \Sigma^*$, find out whether $w$ belongs to the language $L(G)$ generated by $G$, and if so, construct a rightmost derivation of $w$, in a deterministic fashion. Of course, this is not possible for all context-free grammars, but only for those that correspond to languages that can be recognized by a deterministic PDA (DPDA). Knuth’s major discovery was that for a certain type of grammars, the $LR(k)$-grammars, a certain kind of DPDA could be constructed from the grammar (shift/reduce parsers). The $k$ in $LR(k)$ refers to the amount of lookahead that is necessary in order to proceed deterministically. It turns out that $k = 1$ is sufficient, but even in this case, Knuth construction produces very large DPDA’s, and his original $LR(1)$ method is not practical. Fortunately, around 1969, Frank DeRemer, in his MIT Ph.D. thesis, investigated a practical restriction of Knuth’s method, known as $SLR(k)$, and soon after, the $LALR(k)$ method was
discovered. The \textit{SLR}(k) and the \textit{LALR}(k) methods are both based on the construction of the \textit{LR}(0)-characteristic automaton from a grammar \(G\), and we begin by explaining this construction. The additional ingredient needed to obtain an \textit{SLR}(k) or an \textit{LALR}(k) parser from an \textit{LR}(0) parser is the computation of lookahead sets. In the \textit{SLR} case, the FOLLOW sets are needed, and in the \textit{LALR} case, a more sophisticated version of the FOLLOW sets is needed. We will consider the construction of these sets in the case \(k = 1\). We will discuss the shift/reduce algorithm and consider briefly ways of building \textit{LR}(1)-parsing tables.

For simplicity of exposition, we first assume that grammars have no \(\epsilon\)-rules. This restriction will be lifted in Section 4.10. Given a reduced context-free grammar \(G = (V, \Sigma, P, S')\) augmented with start production \(S' \rightarrow S\), where \(S'\) does not appear in any other productions, the set \(C_G\) of \textit{characteristic strings} of \(G\) is the following subset of \(V^*\) (watch out, not \(\Sigma^*\)):

\[
C_G = \{ \alpha \beta \in V^* | S' \xrightarrow{r_m} \alpha B v \xrightarrow{r_m} \alpha \beta v, \alpha, \beta \in V^*, v \in \Sigma^*, B \rightarrow \beta \in P \}.
\]

In words, \(C_G\) is a certain set of prefixes of sentential forms obtained in rightmost derivations: Those obtained by truncating the part of the sentential form immediately following the rightmost symbol in the righthand side of the production applied at the last step.

The fundamental property of LR-parsing, due to D. Knuth, is that \(C_G\) is a \textit{regular language}. Furthermore, a DFA, \(DCG\), accepting \(C_G\), can be constructed from \(G\).

Conceptually, it is simpler to construct the DFA accepting \(C_G\) in two steps:

1. First, construct a nondeterministic automaton with \(\epsilon\)-rules, \(NCG\), accepting \(C_G\).
2. Apply the subset construction (Rabin and Scott’s method) to \(NCG\) to obtain the DFA \(DCG\).

In fact, careful inspection of the two steps of this construction reveals that it is possible to construct \(DCG\) directly in a single step, and this is the construction usually found in most textbooks on parsing.

The nondeterministic automaton \(NCG\) accepting \(C_G\) is defined as follows:

The states of \(NCG\) are “marked productions”, where a marked production is a string of the form \(A \rightarrow \alpha \text{"."} \beta\), where \(A \rightarrow \alpha \beta\) is a production, and “.” is a symbol not in \(V\) called the “dot” and which can appear anywhere within \(\alpha \beta\).

The start state is \(S' \rightarrow \text{"."} S\), and the transitions are defined as follows:

(a) For every terminal \(a \in \Sigma\), if \(A \rightarrow \alpha \text{".”} a \beta\) is a marked production, with \(\alpha, \beta \in V^*\), then there is a transition on input \(a\) from state \(A \rightarrow \alpha \text{".”} a \beta\) to state \(A \rightarrow \alpha a \text{".”} \beta\) obtained by “shifting the dot.” Such a transition is shown in Figure 4.1.
Figure 4.1: Transition on terminal input $a$

Figure 4.2: Transitions from a state $A \rightarrow \alpha \cdot \cdot B \beta$

(b) For every nonterminal $B \in N$, if $A \rightarrow \alpha \cdot \cdot B \beta$ is a marked production, with $\alpha, \beta \in V^*$, then there is a transition on input $B$ from state $A \rightarrow \alpha \cdot \cdot B \beta$ to state $A \rightarrow \alpha B \cdot \cdot \cdot \beta$ (obtained by “shifting the dot”), and transitions on input $\epsilon$ (the empty string) to all states $B \rightarrow \cdot \cdot \gamma_i$, for all productions $B \rightarrow \gamma_i$ with left-hand side $B$. Such transitions are shown in Figure 4.2.

(c) A state is final if and only if it is of the form $A \rightarrow \beta \cdot \cdot \cdot$ (that is, the dot is in the rightmost position).

The above construction is illustrated by the following example:
Example 1. Consider the grammar $G_1$ given by:

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow aEb \\
E & \rightarrow ab
\end{align*}
\]

The NFA for $C_{G_1}$ is shown in Figure 4.3. The result of making the NFA for $C_{G_1}$ deterministic is shown in Figure 4.4 (where transitions to the “dead state” have been omitted). The internal structure of the states 1, \ldots, 6 is shown below:

\[
\begin{align*}
1: S & \rightarrow .E \\
     & \rightarrow .aEb \\
     & \rightarrow .ab \\
2: E & \rightarrow a.Eb \\
     & \rightarrow a.b \\
     & \rightarrow .aEb \\
     & \rightarrow .ab \\
3: E & \rightarrow aE.b \\
4: S & \rightarrow E. \\
5: E & \rightarrow ab. \\
6: E & \rightarrow aEb.
\end{align*}
\]

The next example is slightly more complicated.

Example 2. Consider the grammar $G_2$ given by:

\[
\begin{align*}
S & \rightarrow E \\
E & \rightarrow E + T \\
E & \rightarrow T \\
T & \rightarrow T * a \\
T & \rightarrow a
\end{align*}
\]

The result of making the NFA for $C_{G_2}$ deterministic is shown in Figure 4.5 (where transitions to the “dead state” have been omitted). The internal structure of the states 1, \ldots, 8
4.1. LR(0)-CHARACTERISTIC AUTOMATA

Figure 4.3: NFA for $C_{G_1}$

Figure 4.4: DFA for $C_{G_1}$
Figure 4.5: DFA for $C_{G_2}$

is shown below:

1: $S \rightarrow .E$
   $E \rightarrow .E + T$
   $E \rightarrow .T$
   $T \rightarrow .T * a$
   $T \rightarrow .a$

2: $E \rightarrow E + T$
   $S \rightarrow E.$

3: $E \rightarrow T.$
   $T \rightarrow T * a$

4: $T \rightarrow a.$

5: $E \rightarrow E + T$
   $T \rightarrow T * a$

6: $T \rightarrow T * a.$

7: $E \rightarrow E + T.$
   $T \rightarrow T * a$

8: $T \rightarrow T * a.$

Note that some of the marked productions are more important than others. For example, in state 5, the marked production $E \rightarrow E + .T$ determines the state. The other two items $T \rightarrow .T * a$ and $T \rightarrow .a$ are obtained by $\epsilon$-closure.

We call a marked production of the form $A \rightarrow \alpha . \beta$, where $\alpha \neq \epsilon$, a core item. A marked production of the form $A \rightarrow \beta$, is called a reduce item. Reduce items only appear in final
If we also call $S' \rightarrow .S$ a core item, we observe that every state is completely determined by its subset of core items. The other items in the state are obtained via $\epsilon$-closure. We can take advantage of this fact to write a more efficient algorithm to construct in a single pass the $LR(0)$-automaton.

Also observe the so-called spelling property: All the transitions entering any given state have the same label.

Given a state $s$, if $s$ contains both a reduce item $A \rightarrow \gamma$ and a shift item $B \rightarrow \alpha.a\beta$, where $a \in \Sigma$, we say that there is a shift/reduce conflict in state $s$ on input $a$. If $s$ contains two (distinct) reduce items $A_1 \rightarrow \gamma_1$ and $A_2 \rightarrow \gamma_2$, we say that there is a reduce/reduce conflict in state $s$.

A grammar is said to be $LR(0)$ if the DFA $DCG$ has no conflicts. This is the case for the grammar $G_1$. However, it should be emphasized that this is extremely rare in practice. The grammar $G_1$ is just very nice, and a toy example. In fact, $G_2$ is not $LR(0)$.

To eliminate conflicts, one can either compute $SLR(1)$-lookahead sets, using FOLLOW sets (see Section 4.6), or sharper lookahead sets, the $LALR(1)$ sets (see Section 4.9). For example, the computation of $SLR(1)$-lookahead sets for $G_2$ will eliminate the conflicts.

We will describe methods for computing $SLR(1)$-lookahead sets and $LALR(1)$-lookahead sets in Sections 4.6, 4.9, and 4.10. A more drastic measure is to compute the $LR(1)$-automaton, in which the states incorporate lookahead symbols (see Section 4.11). However, as we said before, this is not a practical methods for large grammars.

In order to motivate the construction of a shift/reduce parser from the DFA accepting $C_G$, let us consider a rightmost derivation for $w = aaabbb$ in reverse order for the grammar

$$
\begin{align*}
0: & \quad S \rightarrow E \\
1: & \quad E \rightarrow aEb \\
2: & \quad E \rightarrow ab \\
\end{align*}
$$

| $aaabbb$   | $\alpha_1\beta_1v_1$    | $E \rightarrow ab$ |
| $aaEbb$    | $\alpha_1B_1v_1$        | $E \rightarrow aEb$ |
| $aaEbb$    | $\alpha_2\beta_2v_2$    | $E \rightarrow aEb$ |
| $aEbb$     | $\alpha_2B_2v_2$        | $E \rightarrow aEb$ |
| $aEbb$     | $\alpha_3\beta_3v_3$    | $\alpha_3 = v_3 = \epsilon$ |
| $E$        | $\alpha_3B_3v_3$        | $\alpha_3 = v_3 = \epsilon$ |
| $E$        | $\alpha_4\beta_4v_4$    | $\alpha_4 = v_4 = \epsilon$ |
| $S$        | $\alpha_4B_4v_4$        | $\alpha_4 = v_4 = \epsilon$ |

$S \rightarrow E$
Observe that the strings $\alpha_i\beta_i$ for $i = 1, 2, 3, 4$ are all accepted by the DFA for $C_G$ shown in Figure 4.6.

Also, every step from $\alpha_i\beta_iv_i$ to $\alpha_iB_iv_i$ is the inverse of the derivation step using the production $B_i \rightarrow \beta_i$, and the marked production $B_i \rightarrow \beta_i \ldots$ is one of the reduce items in the final state reached after processing $\alpha_i\beta_i$ with the DFA for $C_G$.

This suggests that we can parse $w = aaabbb$ by recursively running the DFA for $C_G$.

The first time (which correspond to step 1) we run the DFA for $C_G$ on $w$, some string $\alpha_1\beta_1$ is accepted and the remaining input in $v_1$.

Then, we “reduce” $\beta_1$ to $B_1$ using a production $B_1 \rightarrow \beta_1$ corresponding to some reduce item $B_1 \rightarrow \beta_1 \ldots$ in the final state $s_1$ reached on input $\alpha_1\beta_1$.

We now run the DFA for $C_G$ on input $\alpha_1B_1v_1$. The string $\alpha_2\beta_2$ is accepted, and we have

$$\alpha_1B_1v_1 = \alpha_2\beta_2v_2.$$  

We reduce $\beta_2$ to $B_2$ using a production $B_2 \rightarrow \beta_2$ corresponding to some reduce item $B_2 \rightarrow \beta_2 \ldots$ in the final state $s_2$ reached on input $\alpha_2\beta_2$.

We now run the DFA for $C_G$ on input $\alpha_2B_2v_2$, and so on.

At the $(i+1)$th step $(i \geq 1)$, we run the DFA for $C_G$ on input $\alpha_iB_iv_i$. The string $\alpha_{i+1}\beta_{i+1}$ is accepted, and we have

$$\alpha_iB_iv_i = \alpha_{i+1}\beta_{i+1}v_{i+1}.$$  

We reduce $\beta_{i+1}$ to $B_{i+1}$ using a production $B_{i+1} \rightarrow \beta_{i+1}$ corresponding to some reduce item $B_{i+1} \rightarrow \beta_{i+1} \ldots$ in the final state $s_{i+1}$ reached on input $\alpha_{i+1}\beta_{i+1}$.

The string $\beta_{i+1}$ in $\alpha_{i+1}\beta_{i+1}v_{i+1}$ if often called a handle.

Then we run again the DFA for $C_G$ on input $\alpha_{i+1}B_{i+1}v_{i+1}$.

Now, because the DFA for $C_G$ is deterministic there is no need to rerun it on the entire string $\alpha_{i+1}B_{i+1}v_{i+1}$, because on input $\alpha_{i+1}$ it will take us to the same state, say $p_{i+1}$, that it reached on input $\alpha_{i+1}\beta_{i+1}v_{i+1}$.  

![Figure 4.6: DFA for $C_G$](image-url)
The trick is that we can use a stack to keep track of the sequence of states used to process \( \alpha_{i+1}\beta_{i+1} \).

Then, to perform the reduction of \( \alpha_{i+1}\beta_{i+1} \) to \( \alpha_{i+1}B_{i+1} \), we simply pop a number of states equal to \( |\beta_{i+1}| \), covering a new state \( p_{i+1} \) on top of the stack, and from state \( p_{i+1} \) we perform the transition on input \( B_{i+1} \) to a state \( q_{i+1} \) (in the DFA for \( C_G \)), so we push state \( q_{i+1} \) on the stack which now contains the sequence of states on input \( \alpha_{i+1}B_{i+1} \) that takes us to \( q_{i+1} \).

Then we resume scanning \( v_{i+1} \) using the DGA for \( C_G \), pushing each state being traversed on the stack until we hit a final state.

At this point we find the new string \( \alpha_{i+2}\beta_{i+2} \) that leads to a final state and we continue as before.

The process stops when the remaining input \( v_{i+1} \) becomes empty and when the reduce item \( S' \rightarrow S \) (here, \( S \rightarrow E. \)) belongs to the final state \( s_{i+1} \).

![Figure 4.7: DFA for \( C_G \)](image)

For example, on input \( \alpha_2\beta_2 = aEbb \), we have the sequence of states:

\[
1 \ 2 \ 2 \ 3 \ 6
\]

State 6 contains the marked production \( E \rightarrow aEbb^* \), so we pop the three topmost states 2 3 6 obtaining the stack

\[
1 \ 2
\]

and then we make the transition from state 2 on input \( E \), which takes us to state 3, so we push 3 on top of the stack, obtaining

\[
1 \ 2 \ 3
\]

We continue from state 3 on input \( b \).

Basically, the recursive calls to the DFA for \( C_G \) are implemented using a stack.
What is not clear is, during step $i + 1$, when reaching a final state $s_{i+1}$, how do we know which production $B_{i+1} \rightarrow \beta_{i+1}$ to use in the reduction step?

Indeed, state $s_{i+1}$ could contain several reduce items $B_{i+1} \rightarrow \beta_{i+1}$.

This is where we assume that we were able to compute some lookahead information, that is, for every final state $s$ and every input $a$, we know which unique production $n: B_{i+1} \rightarrow \beta_{i+1}$ applies. This is recorded in a table name “action,” such that action($s$, $a$) = rn, where “r” stands for reduce.

Typically we compute SLR(1) or LALR(1) lookahead sets. Otherwise, we could pick some reducing production nondeterministically and use backtracking. This works but the running time may be exponential.

The DFA for $C_G$ and the action table giving us the reductions can be combined to form a bigger action table which specifies completely how the parser using a stack works.

This kind of parser called a shift-reduce parser is discussed in the next section.

In order to make it easier to compute the reduce entries in the parsing table, we assume that the end of the input $w$ is signalled by a special endmarker traditionally denoted by $\$.$

### 4.2 Shift/Reduce Parsers

A shift/reduce parser is a modified kind of DPDA. Firstly, push moves, called shift moves, are restricted so that exactly one symbol is pushed on top of the stack. Secondly, more powerful kinds of pop moves, called reduce moves, are allowed. During a reduce move, a finite number of stack symbols may be popped off the stack, and the last step of a reduce move, called a goto move, consists of pushing one symbol on top of new topmost symbol in the stack. Shift/reduce parsers use parsing tables constructed from the $LR(0)$-characteristic automaton $DCG$ associated with the grammar. The shift and goto moves come directly from the transition table of $DCG$, but the determination of the reduce moves requires the computation of lookahead sets. The $SLR(1)$ lookahead sets are obtained from some sets called the FOLLOW sets (see Section 4.6), and the $LALR(1)$ lookahead sets $LA(s, A \rightarrow \gamma)$ require fancier FOLLOW sets (see Section 4.9).

The construction of shift/reduce parsers is made simpler by assuming that the end of input strings $w \in \Sigma^*$ is indicated by the presence of an endmarker, usually denoted $\$$, and assumed not to belong to $\Sigma$.

Consider the grammar $G_1$ of Example 1, where we have numbered the productions 0, 1, 2:

$$
0: S \rightarrow E \\
1: E \rightarrow aEb \\
2: E \rightarrow ab
$$

The parsing tables associated with the grammar $G_1$ are shown below:
4.2. SHIFT/REDUCE PARSERS

Entries of the form \( s_i \) are shift actions, where \( i \) denotes one of the states, and entries of the form \( r_n \) are reduce actions, where \( n \) denotes a production number (not a state). The special action acc means accept, and signals the successful completion of the parse. Entries of the form \( i \), in the rightmost column, are goto actions. All blank entries are error entries, and mean that the parse should be aborted.

We will use the notation \( \text{action}(s, a) \) for the entry corresponding to state \( s \) and terminal \( a \in \Sigma \cup \{\$\} \), and \( \text{goto}(s, A) \) for the entry corresponding to state \( s \) and nonterminal \( A \in N - \{S'\} \).

Assuming that the input is \( w\$ \), we now describe in more detail how a shift/reduce parser proceeds. The parser uses a stack in which states are pushed and popped. Initially, the stack contains state 1 and the cursor pointing to the input is positioned on the leftmost symbol. There are four possibilities:

1. If \( \text{action}(s, a) = sj \), then push state \( j \) on top of the stack, and advance to the next input symbol in \( w\$ \). This is a shift move.

2. If \( \text{action}(s, a) = rn \), then do the following: First, determine the length \( k = |\gamma| \) of the righthand side of the production \( n: A \rightarrow \gamma \). Then, pop the topmost \( k \) symbols off the stack (if \( k = 0 \), no symbols are popped). If \( p \) is the new top state on the stack (after the \( k \) pop moves), push the state \( \text{goto}(p, A) \) on top of the stack, where \( A \) is the lefthand side of the “reducing production” \( A \rightarrow \gamma \). Do not advance the cursor in the current input. This is a reduce move.
(3) If action\((s, \$) = \text{acc}\), then accept. The input string \(w\) belongs to \(L(G)\).

(4) In all other cases, \text{error}, abort the parse. The input string \(w\) does not belong to \(L(G)\).

Observe that no explicit state control is needed. The current state is always the current topmost state in the stack. We illustrate below a parse of the input \(aaabbb\$\).

<table>
<thead>
<tr>
<th>stack</th>
<th>remaining input</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>aaabbb$</td>
<td>(s2)</td>
</tr>
<tr>
<td>12</td>
<td>aaabbb$</td>
<td>(s2)</td>
</tr>
<tr>
<td>122</td>
<td>abbb$</td>
<td>(s2)</td>
</tr>
<tr>
<td>1222</td>
<td>bbb$</td>
<td>(s5)</td>
</tr>
<tr>
<td>12225</td>
<td>bb$</td>
<td>(r2)</td>
</tr>
<tr>
<td>1223</td>
<td>bb$</td>
<td>(s6)</td>
</tr>
<tr>
<td>12236</td>
<td>b$</td>
<td>(r1)</td>
</tr>
<tr>
<td>123</td>
<td>b$</td>
<td>(s6)</td>
</tr>
<tr>
<td>1236</td>
<td>$</td>
<td>(r1)</td>
</tr>
<tr>
<td>14</td>
<td>$</td>
<td>\text{acc}</td>
</tr>
</tbody>
</table>

Observe that the sequence of reductions read from bottom-up yields a rightmost derivation of \(aaabbb\) from \(E\) (or from \(S\), if we view the action acc as the reduction by the production \(S \rightarrow E\)). This is a general property of \(LR\)-parsers.

The \(SLR(1)\) reduce entries in the parsing tables are determined as follows: For every state \(s\) containing a reduce item \(B \rightarrow \gamma\), if \(B \rightarrow \gamma\) is the production number \(n\), enter the action \(rn\) for state \(s\) and every terminal \(a \in \text{FOLLOW}(B)\). If the resulting shift/reduce parser has no conflicts, we say that the grammar is \(SLR(1)\). For the \(LALR(1)\) reduce entries, enter the action \(rn\) for state \(s\) and production \(n: B \rightarrow \gamma\), for all \(a \in \text{LA}(s, B \rightarrow \gamma)\). Similarly, if the shift/reduce parser obtained using \(LALR(1)\)-lookahead sets has no conflicts, we say that the grammar is \(LALR(1)\).

### 4.3 Computation of FIRST

In order to compute the FOLLOW sets, we first need to to compute the FIRST sets! For simplicity of exposition, we first assume that grammars have no \(\epsilon\)-rules. The general case will be treated in Section 4.10.

Given a context-free grammar \(G = (V, \Sigma, P, S')\) (augmented with a start production \(S' \rightarrow S\)), for every nonterminal \(A \in N = V - \Sigma\), let

\[
\text{FIRST}(A) = \{ a \mid a \in \Sigma, A \xrightarrow{\alpha} aa, \text{ for some } \alpha \in V^* \}.
\]

For a terminal \(a \in \Sigma\), let \(\text{FIRST}(a) = \{ a \}\). The key to the computation of \(\text{FIRST}(A)\) is the following observation: \(a\) is in \(\text{FIRST}(A)\) if either \(a\) is in

\[
\text{INITFIRST}(A) = \{ a \mid a \in \Sigma, A \rightarrow a\alpha \in P, \text{ for some } \alpha \in V^* \};
\]
or $a$ is in

\[ \{ a \mid a \in \text{FIRST}(B), A \rightarrow B\alpha \in P, \text{ for some } \alpha \in V^*, B \neq A \}. \]

Note that the second assertion is true because, if $B \xrightarrow{\delta} a\delta$, then $A \xrightarrow{\alpha} B\alpha \xrightarrow{\delta} a\delta\alpha$, and so, $\text{FIRST}(B) \subseteq \text{FIRST}(A)$ whenever $A \rightarrow B\alpha \in P$, with $A \neq B$. Hence, the FIRST sets are the least solution of the following set of recursive equations: For each nonterminal $A$,

\[ \text{FIRST}(A) = \text{INITFIRST}(A) \cup \bigcup \{ \text{FIRST}(B) \mid A \rightarrow B\alpha \in P, A \neq B \}. \]

In order to explain the method for solving such systems, we will formulate the problem in more general terms, but first, we describe a “naive” version of the shift/reduce algorithm that hopefully demystifies the “optimized version” described in Section 4.2.

### 4.4 The Intuition Behind the Shift/Reduce Algorithm

Let $DCG = (K, V, \delta, q_0, F)$ be the DFA accepting the regular language $C_G$, and let $\delta^*$ be the extension of $\delta$ to $K \times V^*$. Let us assume that the grammar $G$ is either SLR(1) or LALR(1), which implies that it has no shift/reduce or reduce/reduce conflicts. We can use the DFA $DCG$ accepting $C_G$ recursively to parse $L(G)$. The function $CG$ is defined as follows: Given any string $\mu \in V^*$,

\[
CG(\mu) = \begin{cases} 
\text{error} & \text{if } \delta^*(q_0, \mu) = \text{error}; \\
(\delta^*(q_0, \theta), \theta, v) & \text{if } \delta^*(q_0, \theta) \in F, \mu = \theta v \text{ and } \theta \text{ is the shortest prefix of } \mu \text{ s.t. } \delta^*(q_0, \theta) \in F.
\end{cases}
\]

The naive shift-reduce algorithm is shown below:

begin
accept := true;
stop := false;
$\mu := w$; \{input string\}
while ¬stop do
if $CG(\mu) = \text{error}$ then
stop := true; accept := false
else
Let $(q, \theta, v) = CG(\mu)$
Let $B \rightarrow \beta$ be the production so that
action$(q, \text{FIRST}(v)) = B \rightarrow \beta$ and let $\theta = \alpha\beta$
if $B \rightarrow \beta = S' \rightarrow S$ then
stop := true
else
\[ \mu := \alpha Bv \quad \text{\{reduction\}} \]

\[ \text{endif} \]

\[ \text{endif} \]

\[ \text{endwhile} \]

\[ \text{end} \]

The idea is to recursively run the DFA \( DCG \) on the sentential form \( \mu \), until the first final state \( q \) is hit. Then, the sentential form \( \mu \) must be of the form \( \alpha \beta v \), where \( v \) is a terminal string ending in \$, and the final state \( q \) contains a reduce item of the form \( B \rightarrow \beta \), with \( \text{action}(q, \text{FIRST}(v)) = B \rightarrow \beta \). Thus, we can reduce \( \mu = \alpha \beta v \) to \( \alpha Bv \), since we have found a rightmost derivation step, and repeat the process.

Note that the major inefficiency of the algorithm is that when a reduction is performed, the prefix \( \alpha \) of \( \mu \) is reparsed entirely by \( DCG \). Since \( DCG \) is deterministic, the sequence of states obtained on input \( \alpha \) is uniquely determined. If we keep the sequence of states produced on input \( \theta \) by \( DCG \) in a stack, then it is possible to avoid reparsing \( \alpha \). Indeed, all we have to do is update the stack so that just before applying \( DCG \) to \( \alpha Av \), the sequence of states in the stack is the sequence obtained after parsing \( \alpha \). This stack is obtained by popping the \( |\beta| \) topmost states and performing an update which is just a goto move. This is the standard version of the shift/reduce algorithm!

### 4.5 The Graph Method for Computing Fixed Points

Let \( X \) be a finite set representing the domain of the problem (in Section 4.3 above, \( X = \Sigma \)), let \( F(1), \ldots, F(N) \) be \( N \) sets to be computed and let \( I(1), \ldots, I(N) \) be \( N \) given subsets of \( X \). The sets \( I(1), \ldots, I(N) \) are the initial sets. We also have a directed graph \( G \) whose set of nodes is \( \{1, \ldots, N\} \) and which represents relationships among the sets \( F(i) \), where \( 1 \leq i \leq N \). The graph \( G \) has no parallel edges and no loops, but it may have cycles. If there is an edge from \( i \) to \( j \), this is denoted by \( iGj \) (note that the absence of loops means that \( iGi \) never holds). Also, the existence of a path from \( i \) to \( j \) is denoted by \( iG^+j \). The graph \( G \) represents a relation, and \( G^+ \) is the graph of the transitive closure of this relation. The existence of a path from \( i \) to \( j \), including the null path, is denoted by \( iG^*j \). Hence, \( G^* \) is the reflexive and transitive closure of \( G \). We want to solve for the least solution of the system of recursive equations:

\[ F(i) = I(i) \cup \{ F(j) \mid iGj, i \neq j \}, \quad 1 \leq i \leq N. \]

Since \( (2^X)^N \) is a complete lattice under the inclusion ordering (which means that every family of subsets has a least upper bound, namely, the union of this family), it is an \( \omega \)-complete poset, and since the function \( F : (2^X)^N \to (2^X)^N \) induced by the system of equations is easily seen to preserve least upper bounds of \( \omega \)-chains, the least solution of the system can be computed by the standard fixed point technique (as explained in Section 3.7
of the class notes). We simply compute the sequence of approximations \((F^k(1), \ldots, F^k(N))\), where

\[ F^0(i) = \emptyset, \quad 1 \leq i \leq N, \]

and

\[ F^{k+1}(i) = I(i) \cup \bigcup \{ F^k(j) \mid i \in G_j, i \neq j \}, \quad 1 \leq i \leq N. \]

It is easily seen that we can stop at \(k = N - 1\), and the least solution is given by

\[ F(i) = F^1(i) \cup F^2(i) \cup \cdots \cup F^N(i), \quad 1 \leq i \leq N. \]

However, the above expression can be simplified to

\[ F(i) = \bigcup \{ I(j) \mid i \in G^* j \}, \quad 1 \leq i \leq N. \]

This last expression shows that in order to compute \(F(i)\), it is necessary to compute the union of all the initial sets \(I(j)\) reachable from \(i\) (including \(i\)). Hence, any transitive closure algorithm or graph traversal algorithm will do. For simplicity and for pedagogical reasons, we use a depth-first search algorithm.

Going back to FIRST, we see that all we have to do is to compute the INITFIRST sets, the graph \(G_{FIRST}\), and then use the graph traversal algorithm. The graph \(G_{FIRST}\) is computed as follows: The nodes are the nonterminals and there is an edge from \(A\) to \(B\) (\(A \neq B\)) if and only if there is a production of the form \(A \rightarrow B\alpha\), for some \(\alpha \in V^*\).

**Example 1.** Computation of the FIRST sets for the grammar \(G_1\) given by the rules:

\[
S \rightarrow E$
\]
\[
E \rightarrow E + T
\]
\[
E \rightarrow T
\]
\[
T \rightarrow T * F
\]
\[
T \rightarrow F
\]
\[
F \rightarrow (E)
\]
\[
F \rightarrow -T
\]
\[
F \rightarrow a.
\]

We get

\[
INITFIRST(E) = \emptyset, \quad INITFIRST(T) = \emptyset, \quad INITFIRST(F) = \{(, -, a)\}.
\]

The graph \(G_{FIRST}\) is shown in Figure 4.9.

We obtain the following FIRST sets:

\[
FIRST(E) = FIRST(T) = FIRST(F) = \{(, -, a)\}.
\]
4.6 Computation of FOLLOW

Recall the definition of FOLLOW($A$) for a nonterminal $A$:

\[
\text{FOLLOW}(A) = \{ a \mid a \in \Sigma, S \xrightarrow{\alpha} \alpha Aa\beta, \text{ for some } \alpha, \beta \in V^* \}.
\]

Note that $a$ is in FOLLOW($A$) if either $a$ is in 
\[
\text{INITFOLLOW}(A) = \{ a \mid a \in \Sigma, B \rightarrow \alpha AX\beta \in P, a \in \text{FIRST}(X), \alpha, \beta \in V^* \}
\]
or $a$ is in 
\[
\{ a \mid a \in \text{FOLLOW}(B), B \rightarrow \alpha A \in P, \alpha \in V^*, A \neq B \}.
\]

Indeed, if $S \xrightarrow{\lambda} \lambda B a \rho$, then $S \xrightarrow{\lambda} \lambda B a \rho \Rightarrow \lambda \alpha A a \rho$, and so,

\[
\text{FOLLOW}(B) \subseteq \text{FOLLOW}(A)
\]
whenever $B \rightarrow \alpha A$ is in $P$, with $A \neq B$. Hence, the FOLLOW sets are the least solution of the set of recursive equations: For all nonterminals $A$,

\[
\text{FOLLOW}(A) = \text{INITFOLLOW}(A) \cup \bigcup \{ \text{FOLLOW}(B) \mid B \rightarrow \alpha A \in P, \alpha \in V^*, A \neq B \}.
\]

According to the method explained above, we just have to compute the INITFOLLOW sets (using FIRST) and the graph GFOllow, which is computed as follows: The nodes are the nonterminals and there is an edge from $A$ to $B$ ($A \neq B$) if and only if there is a production of the form $B \rightarrow \alpha A$ in $P$, for some $\alpha \in V^*$. Note the duality between the construction of the graph GFIRST and the graph GFOllow.

**Example 2.** Computation of the FOLLOW sets for the grammar $G_1$.

INITFOLLOW($E$) = \{+, \}, INITFOLLOW($T$) = \{\} , INITFOLLOW($F$) = \{\}.

The graph GFOllow is shown in Figure 4.10. We have

\[
\begin{align*}
\text{FOLLOW}(E) &= \text{INITFOLLOW}(E), \\
\text{FOLLOW}(T) &= \text{INITFOLLOW}(T) \cup \text{INITFOLLOW}(E) \cup \text{INITFOLLOW}(F), \\
\text{FOLLOW}(F) &= \text{INITFOLLOW}(F) \cup \text{INITFOLLOW}(T) \cup \text{INITFOLLOW}(E),
\end{align*}
\]

and so

\[
\begin{align*}
\text{FOLLOW}(E) &= \{+, \}, \text{FOLLOW}(T) = \{+, \}, \text{FOLLOW}(F) = \{+, \}.
\end{align*}
\]
4.7 Algorithm Traverse

The input is a directed graph $G_r$ having $N$ nodes, and a family of initial sets $I[i]$, $1 \leq i \leq N$. We assume that a function $\text{successors}$ is available, which returns for each node $n$ in the graph, the list $\text{successors}[n]$ of all immediate successors of $n$. The output is the list of sets $F[i]$, $1 \leq i \leq N$, solution of the system of recursive equations of Section 4.5. Hence,

$$F[i] = \bigcup \{I[j] \mid iG^*j\}, \quad 1 \leq i \leq N.$$

The procedure $\text{Reachable}$ visits all nodes reachable from a given node. It uses a stack $\text{STACK}$ and a boolean array $\text{VISITED}$ to keep track of which nodes have been visited. The procedures $\text{Reachable}$ and $\text{traverse}$ are shown in Figure 4.11.

4.8 More on $LR(0)$-Characteristic Automata

Let $G = (V, \Sigma, P, S')$ be an augmented context-free grammar with augmented start production $S' \longrightarrow SS$ (where $S'$ only occurs in the augmented production). The rightmost derivation relation is denoted by $\Rightarrow_{rm}$.

Recall that the set $C_G$ of characteristic strings for the grammar $G$ is defined by

$$C_G = \{\alpha\beta \in V^* \mid S' \xrightarrow{\ast} \alpha Av \Rightarrow_{rm} \alpha\beta v, \alpha\beta \in V^*, v \in \Sigma^*\}.$$

The fundamental property of LR-parsing, due to D. Knuth, is stated in the following theorem:

**Theorem 4.1.** Let $G$ be a context-free grammar and assume that every nonterminal derives some terminal string. The language $C_G$ (over $V^*$) is a regular language. Furthermore, a deterministic automaton $\text{DCG}$ accepting $C_G$ can be constructed from $G$.

The construction of $\text{DCG}$ can be found in various places, including the book on Compilers by Aho, Sethi and Ullman. We explained this construction in Section 4.1. The proof that the
Procedure Reachable(Gr : graph; startnode : node; I : listof sets;
                 var F : listof sets);

var currentnode, succnode, i : node; STACK : stack;
  VISITED : array[1..N] of boolean;

begin
  for i := 1 to N do
    VISITED[i] := false;
  STACK := EMPTY;
  push(STACK, startnode);
  while STACK ≠ EMPTY do
    begin
      currentnode := top(STACK); pop(STACK);
      VISITED[currentnode] := true;
      for each succnode ∈ successors(currentnode) do
        if ¬VISITED[succnode] then
          begin
            push(STACK, succnode);
            F[startnode] := F[startnode] ∪ I[succnode]
          end
        end
      end
  end

The sets $F[i]$, $1 ≤ i ≤ N$, are computed as follows:

begin
  for i := 1 to N do
    F[i] := I[i];
  for startnode := 1 to N do
    Reachable(Gr, startnode, I, F)
end

Figure 4.11: Algorithm traverse
4.9. LALR(1)-LOOKAHEAD SETS

NFA $NCG$ constructed as indicated in Section 4.1 is correct, i.e., that it accepts precisely $C_G$, is nontrivial, but not really hard either. This will be the object of a homework assignment! However, note a subtle point: The construction of $NCG$ is only correct under the assumption that every nonterminal derives some terminal string. Otherwise, the construction could yield an NFA $NCG$ accepting strings not in $C_G$.

Recall that the states of the characteristic automaton $CGA$ are sets of items (or marked productions), where an item is a production with a dot anywhere in its right-hand side. Note that in constructing $CGA$, it is not necessary to include the state $\{S' \rightarrow S$.} (the endmarker $\$ is only needed to compute the lookahead sets). If a state $p$ contains a marked production of the form $A \rightarrow \beta$, where the dot is the rightmost symbol, state $p$ is called a reduce state and $A \rightarrow \beta$ is called a reducing production for $p$. Given any state $q$, we say that a string $\beta \in V^*$ accesses $q$ if there is a path from some state $p$ to the state $q$ on input $\beta$ in the automaton $CGA$. Given any two states $p, q \in CGA$, for any $\beta \in V^*$, if there is a sequence of transitions in $CGA$ from $p$ to $q$ on input $\beta$, this is denoted by

$$p \xrightarrow{\beta} q.$$ 

The initial state which is the closure of the item $S' \rightarrow .S$ is denoted by 1. The LALR(1)-lookahead sets are defined in the next section.

4.9 LALR(1)-Lookahead Sets

For any reduce state $q$ and any reducing production $A \rightarrow \beta$ for $q$, let

$$\text{LA}(q, A \rightarrow \beta) = \{ a \mid a \in \Sigma, S' \xrightarrow{\ast} \alpha A a v \Rightarrow \alpha \beta a v, \alpha, \beta \in V^*, v \in \Sigma^*, \alpha \beta \text{ accesses } q \}.$$ 

In words, $\text{LA}(q, A \rightarrow \beta)$ consists of the terminal symbols for which the reduction by production $A \rightarrow \beta$ in state $q$ is the correct action (that is, for which the parse will terminate successfully). The LA sets can be computed using the FOLLOW sets defined below.

For any state $p$ and any nonterminal $A$, let

$$\text{FOLLOW}(p, A) = \{ a \mid a \in \Sigma, S' \xrightarrow{\ast} \alpha A a v, \alpha \in V^*, v \in \Sigma^* \text{ and } \alpha \text{ accesses } p \}.$$ 

Since for any derivation

$$S' \xrightarrow{\ast} \alpha A a v \Rightarrow \alpha \beta a v$$

where $\alpha \beta$ accesses $q$, there is a state $p$ such that $p \xrightarrow{\beta} q$ and $\alpha$ accesses $p$, it is easy to see that the following result holds:

**Proposition 4.2.** For every reduce state $q$ and any reducing production $A \rightarrow \beta$ for $q$, we have

$$\text{LA}(q, A \rightarrow \beta) = \bigcup \{ \text{FOLLOW}(p, A) \mid p \xrightarrow{\beta} q \}.$$
Also, we let
\[ \text{LA}(\{S' \rightarrow S.\}, S' \rightarrow S\$) = \text{FOLLOW}(1, S). \]

Intuitively, when the parser makes the reduction by production \( A \rightarrow \beta \) in state \( q \), each state \( p \) as above is a possible top of stack after the states corresponding to \( \beta \) are popped. Then the parser must read \( A \) in state \( p \), and the next input symbol will be one of the symbols in \( \text{FOLLOW}(p, A) \).

The computation of \( \text{FOLLOW}(p, A) \) is similar to that of \( \text{FOLLOW}(A) \). First, we compute \( \text{INITFOLLOW}(p, A) \), given by
\[
\text{INITFOLLOW}(p, A) = \{ a \mid a \in \Sigma, \exists q, r, p \xrightarrow{A} q \xrightarrow{a} r \}. 
\]

These are the terminals that can be read in \( CGA \) after the “goto transition” on nonterminal \( A \) has been performed from \( p \). These sets can be easily computed from \( CGA \).

Note that for the state \( p \) whose core item is \( S' \rightarrow S.\$ \), we have
\[ \text{INITFOLLOW}(p, S) = \{ \$ \}. \]

Next, observe that if \( B \rightarrow \alpha A \) is a production and if
\[ S' \xrightarrow{* \text{rm}} \lambda B a v \]
where \( \lambda \) accesses \( p' \), then
\[ S' \xrightarrow{\text{rm}} \lambda B a v \xrightarrow{\text{rm}} \lambda \alpha A a v \]
where \( \lambda \) accesses \( p' \) and \( p' \xrightarrow{\alpha} p \). Hence \( \lambda \alpha \) accesses \( p \) and
\[ \text{FOLLOW}(p', B) \subseteq \text{FOLLOW}(p, A) \]
whenever there is a production \( B \rightarrow \alpha A \) and \( p' \xrightarrow{\alpha} p \). From this, the following recursive equations are easily obtained: For all \( p \) and all \( A \),
\[
\text{FOLLOW}(p, A) = \text{INITFOLLOW}(p, A) \cup \bigcup \{ \text{FOLLOW}(p', B) \mid B \rightarrow \alpha A \in P, \alpha \in V^* \text{ and } p' \xrightarrow{\alpha} p \}. 
\]

From Section 4.5, we know that these sets can be computed by using the algorithm \textit{traverse}. All we need is to compute the graph \( GLA \).

The nodes of the graph \( GLA \) are the pairs \((p, A)\), where \( p \) is a state and \( A \) is a nonterminal. There is an edge from \((p, A)\) to \((p', B)\) if and only if there is a production of the form \( B \rightarrow \alpha A \) in \( P \) for some \( \alpha \in V^* \) and \( p' \xrightarrow{\alpha} p \) in \( CGA \). Note that it is only necessary to consider nodes \((p, A)\) for which there is a nonterminal transition on \( A \) from \( p \). Such pairs can be obtained from the parsing table. Also, using the \textit{spelling property}, that is, the fact
that all transitions entering a given state have the same label, it is possible to compute the relation \textit{lookback} defined as follows:

\[(q, A) \text{ lookback } (p, A) \iff p \xrightarrow{\beta} q\]

for some reduce state \(q\) and reducing production \(A \rightarrow \beta\). The above considerations show that the FOLLOW sets of Section 4.6 are obtained by ignoring the state component from \(\text{FOLLOW}(p, A)\). We now consider the changes that have to be made when \(\epsilon\)-rules are allowed.

### 4.10 Computing FIRST, FOLLOW, etc. in the Presence of \(\epsilon\)-Rules

[Computing FIRST, FOLLOW and \(\text{LA}(q, A \rightarrow \beta)\) in the Presence of \(\epsilon\)-Rules] First, it is necessary to compute the set \(E\) of erasable nonterminals, that is, the set of nonterminals \(A\) such that \(A \xrightarrow{\dagger} \epsilon\).

We let \(E\) be a boolean array and \(\text{change}\) be a boolean flag. An algorithm for computing \(E\) is shown in Figure 4.12. Then, in order to compute FIRST, we compute

\[
\text{INITFIRST}(A) = \{a \mid a \in \Sigma, A \rightarrow a\alpha \in P, \text{ or } A \rightarrow A_1 \cdots A_k a\alpha \in P, \text{ for some } \alpha \in V^*, \text{ and } E(A_1) = \cdots = E(A_k) = \text{true}\}.
\]

The graph \(G_{\text{FIRST}}\) is obtained as follows: The nodes are the nonterminals, and there is an edge from \(A\) to \(B\) if and only if either there is a production \(A \rightarrow B\alpha\), or a production \(A \rightarrow A_1 \cdots A_k B\alpha\), for some \(\alpha \in V^*\), with \(E(A_1) = \cdots = E(A_k) = \text{true}\). Then, we extend FIRST to strings in \(V^+\), in the obvious way. Given any string \(\alpha \in V^+\), if \(|\alpha| = 1\), then \(\beta = X\) for some \(X \in V\), and

\[
\text{FIRST}(\beta) = \text{FIRST}(X)
\]

as before, else if \(\beta = X_1 \cdots X_n\) with \(n \geq 2\) and \(X_i \in V\), then

\[
\text{FIRST}(\beta) = \text{FIRST}(X_1) \cup \cdots \cup \text{FIRST}(X_k),
\]

where \(k, 1 \leq k \leq n\), is the largest integer so that

\[
E(X_1) = \cdots = E(X_k) = \text{true}.
\]

To compute FOLLOW, we first compute

\[
\text{INITFOLLOW}(A) = \{a \mid a \in \Sigma, B \rightarrow \alpha A\beta \in P, \alpha \in V^*, \beta \in V^+, \text{ and } a \in \text{FIRST}(\beta)\}.
\]

The graph \(G_{\text{FOLLOW}}\) is computed as follows: The nodes are the nonterminals. There is an edge from \(A\) to \(B\) if either there is a production of the form \(B \rightarrow \alpha A\), or \(B \rightarrow \alpha AA_1 \cdots A_k\), for some \(\alpha \in V^*\), and with \(E(A_1) = \cdots = E(A_k) = \text{true}\).
begin
    for each nonterminal $A$ do
        $E(A) := \text{false}$;
    for each nonterminal $A$ such that $A \rightarrow \epsilon \in P$ do
        $E(A) := \text{true}$;
    $change := \text{true}$;
    while $change$ do
        begin
            $change := \text{false}$;
            for each $A \rightarrow A_1 \cdots A_n \in P$
                s.t. $E(A_1) = \cdots = E(A_n) = \text{true}$ do
                    if $E(A) = \text{false}$ then
                        begin
                            $E(A) := \text{true}$;
                            $change := \text{true}$
                        end
        end
end

Figure 4.12: Algorithm for computing $E$
The computation of the LALR(1) lookahead sets is also more complicated because another graph is needed in order to compute INITFOLLOW$(p,A)$. First, the graph $GLA$ is defined in the following way: The nodes are still the pairs $(p, A)$, as before, but there is an edge from $(p, A)$ to $(p', B)$ if and only if either there is some production $B \rightarrow \alpha A$, for some $\alpha \in V^*$ and $p' \xrightarrow{\alpha} p$, or a production $B \rightarrow \alpha A\beta$, for some $\alpha \in V^*$, $\beta \in V^+$, $\beta \xrightarrow{\dagger} \epsilon$, and $p' \xrightarrow{\alpha} p$. The sets INITFOLLOW$(p,A)$ are computed in the following way: First, let

$$DR(p,A) = \{ a \mid a \in \Sigma, \exists q, r, p \xrightarrow{A} q \xrightarrow{\alpha} r \}.$$  

The sets $DR(p,A)$ are the direct read sets. Note that for the state $p$ whose core item is $S' \rightarrow S.\$, we have

$$DR(p,S) = \{ S \}.$$  

Then,

$$INITFOLLOW(p, A) = DR(p, A) \cup \bigcup \{ a \mid a \in \Sigma, S' \xrightarrow{rm} \alpha A\beta a v \xrightarrow{rm} \alpha A a v, \alpha \in V^*, \beta \in V^+, \beta \xrightarrow{\dagger} \epsilon, \alpha \text{ accesses } p \}.$$  

The set INITFOLLOW$(p,A)$ is the set of terminals that can be read before any handle containing $A$ is reduced. The graph $GREAD$ is defined as follows: The nodes are the pairs $(p, A)$, and there is an edge from $(p, A)$ to $(r, C)$ if and only if $p \xrightarrow{A} r$ and $r \xrightarrow{C} s$, for some $s$, with $E(C) = \text{true}$.

Then, it is not difficult to show that the INITFOLLOW sets are the least solution of the set of recursive equations:

$$INITFOLLOW(p, A) = DR(p, A) \cup \bigcup \{ INITFOLLOW(r, C) \mid (p, A) GREAD (r, C) \}.$$  

Hence the INITFOLLOW sets can be computed using the algorithm traverse on the graph $GREAD$ and the sets $DR(p,A)$, and then, the FOLLOW sets can be computed using traverse again, with the graph $GLA$ and sets INITFOLLOW. Finally, the sets LA$(q,A \rightarrow \beta)$ are computed from the FOLLOW sets using the graph lookback.

From section 4.5, we note that $F(i) = F(j)$ whenever there is a path from $i$ to $j$ and a path from $j$ to $i$, that is, whenever $i$ and $j$ are strongly connected. Hence, the solution of the system of recursive equations can be computed more efficiently by finding the maximal strongly connected components of the graph $G$, since $F$ has a same value on each strongly connected component. This is the approach followed by DeRemer and Pennello in: Efficient Computation of LALR(1) Lookahead sets, by F. DeRemer and T. Pennello, TOPLAS, Vol. 4, No. 4, October 1982, pp. 615-649.

We now give an example of grammar which is LALR(1) but not SLR(1).
Example 3. The grammar $G_2$ is given by:

$$
\begin{align*}
S' & \rightarrow S$ \\
S & \rightarrow L = R \\
S & \rightarrow R \\
L & \rightarrow *R \\
L & \rightarrow id \\
R & \rightarrow L
\end{align*}
$$

The states of the characteristic automaton $CGA_2$ are:

1: $S' \rightarrow .S$

$$
\begin{align*}
S & \rightarrow .L = R \\
S & \rightarrow .R \\
L & \rightarrow .*R \\
L & \rightarrow .id \\
R & \rightarrow .L
\end{align*}
$$

2: $S' \rightarrow S.$

3: $S \rightarrow L = R$

$$
\begin{align*}
R & \rightarrow L
\end{align*}
$$

4: $S \rightarrow R.$

5: $L \rightarrow .R$

$$
\begin{align*}
R & \rightarrow .L \\
L & \rightarrow .*R \\
L & \rightarrow .id
\end{align*}
$$

6: $L \rightarrow id.$

7: $S \rightarrow L = .R$

$$
\begin{align*}
R & \rightarrow .L \\
L & \rightarrow .*R \\
L & \rightarrow .id
\end{align*}
$$

8: $L \rightarrow *R.$

9: $R \rightarrow L.$

10: $S \rightarrow L = R.$

We find that

$$
\begin{align*}
\text{INITFIRST}(S) &= \emptyset \\
\text{INITFIRST}(L) &= \{*, id\} \\
\text{INITFIRST}(R) &= \emptyset.
\end{align*}
$$
4.10. COMPUTING FIRST, FOLLOW, ETC. IN THE PRESENCE OF $e$-RULES

The graph $G_{FIRST}$ is shown in Figure 4.13.

Then, we find that

$$\text{FIRST}(S) = \{*, id\}$$
$$\text{FIRST}(L) = \{*, id\}$$
$$\text{FIRST}(R) = \{*, id\}.$$

We also have

$$\text{INITFOLLOW}(S) = \{\$$
$$\text{INITFOLLOW}(L) = \{=\}$$
$$\text{INITFOLLOW}(R) = \emptyset.$$

The graph $G_{FOLLOW}$ is shown in Figure 4.14.

Then, we find that

$$\text{FOLLOW}(S) = \{\$$
$$\text{FOLLOW}(L) = \{=, \$$
$$\text{FOLLOW}(R) = \{=, \$$

Note that there is a shift/reduce conflict in state 3 on input $=$, since there is a shift on input $=$ (since $S \rightarrow L. = R$ is in state 3), and a reduce for $R \rightarrow L$, since $=$ is in
FOLLOW($R$). However, as we shall see, the conflict is resolved if the LALR(1) lookahead sets are computed.

The graph $GLA$ is shown in Figure 4.15.

We get the following INITFOLLOW and FOLLOW sets:

\[
\begin{align*}
\text{INITFOLLOW}(1, S) & = \{\$\} & \text{INITFOLLOW}(1, S) & = \{\$\} \\
\text{INITFOLLOW}(1, R) & = \emptyset & \text{INITFOLLOW}(1, R) & = \{\$\} \\
\text{INITFOLLOW}(1, L) & = \{=\} & \text{INITFOLLOW}(1, L) & = \{=, \$\} \\
\text{INITFOLLOW}(5, R) & = \emptyset & \text{INITFOLLOW}(5, R) & = \{=, \$\} \\
\text{INITFOLLOW}(5, L) & = \emptyset & \text{INITFOLLOW}(5, L) & = \{=, \$\} \\
\text{INITFOLLOW}(7, R) & = \emptyset & \text{INITFOLLOW}(7, R) & = \{\$\} \\
\text{INITFOLLOW}(7, L) & = \emptyset & \text{INITFOLLOW}(7, L) & = \{\$\}.
\end{align*}
\]

Thus, we get

\[
\begin{align*}
\text{LA}(2, S' \rightarrow S\$) & = \text{FOLLOW}(1, S) = \{\$\} \\
\text{LA}(3, R \rightarrow L) & = \text{FOLLOW}(1, R) = \{\$\} \\
\text{LA}(4, S \rightarrow R) & = \text{FOLLOW}(1, S) = \{\$\} \\
\text{LA}(6, L \rightarrow \text{id}) & = \text{FOLLOW}(1, L) \cup \text{FOLLOW}(5, L) \cup \text{FOLLOW}(7, L) = \{=, \$\} \\
\text{LA}(8, L \rightarrow *R) & = \text{FOLLOW}(1, L) \cup \text{FOLLOW}(5, L) \cup \text{FOLLOW}(7, L) = \{=, \$\} \\
\text{LA}(9, R \rightarrow L) & = \text{FOLLOW}(5, R) \cup \text{FOLLOW}(7, R) = \{=, \$\} \\
\text{LA}(10, S \rightarrow L = R) & = \text{FOLLOW}(1, S) = \{\$\}.
\end{align*}
\]

Since LA(3, R → L) does not contain =, the conflict is resolved.
4.11  LR(1)-Characteristic Automata

We conclude this brief survey on LR-parsing by describing the construction of LR(1)-parsers. The new ingredient is that when we construct an NFA accepting \( C_G \), we incorporate lookahead symbols into the states. Thus, a state is a pair \( (A \rightarrow \alpha \beta, b) \), where \( A \rightarrow \alpha \beta \) is a marked production, as before, and \( b \in \Sigma \cup \{\$\} \) is a lookahead symbol. The new twist in the construction of the nondeterministic characteristic automaton is the following:

The start state is \( (S' \rightarrow .S, \$) \), and the transitions are defined as follows:

(a) For every terminal \( a \in \Sigma \), then there is a transition on input \( a \) from state \( (A \rightarrow \alpha \beta, b) \) to the state \( (A \rightarrow \alpha a \beta, b) \) obtained by “shifting the dot” (where \( a = b \) is possible). Such a transition is shown in Figure 4.16.

(b) For every nonterminal \( B \in N \), there is a transition on input \( B \) from state \( (A \rightarrow \alpha B \beta, b) \) to state \( (A \rightarrow \alpha B . \beta, b) \) (obtained by “shifting the dot”), and transitions on input \( \epsilon \) (the empty string) to all states \( (B \rightarrow .\gamma, a) \), for all productions \( B \rightarrow \gamma \) with left-hand side \( B \) and all \( a \in \text{FIRST}(\beta b) \). Such transitions are shown in Figure 4.17.

(c) A state is final if and only if it is of the form \( (A \rightarrow \beta ., b) \) (that is, the dot is in the rightmost position).

Example 3. Consider the grammar \( G_3 \) given by:

\[
\begin{align*}
0: & \quad S \rightarrow E \\
1: & \quad E \rightarrow aEb \\
2: & \quad E \rightarrow \epsilon
\end{align*}
\]
The result of making the NFA for $C_{G_3}$ deterministic is shown in Figure 4.18 (where transitions to the “dead state” have been omitted). The internal structure of the states 1, ..., 8 is shown below:

1: $S \rightarrow .E, \$ 
   $E \rightarrow .aEb, \$
   $E \rightarrow ., \$

2: $E \rightarrow a.Eb, \$
   $E \rightarrow .aEb, b$
   $E \rightarrow ., b$

3: $E \rightarrow a.Eb, b$
   $E \rightarrow .aEb, b$
   $E \rightarrow ., b$

4: $E \rightarrow aEb, \$

5: $E \rightarrow aEb, \$
   $E \rightarrow aE.b, b$

6: $E \rightarrow aE.b, b$

7: $E \rightarrow aEb, b$

8: $S \rightarrow E., \$

The $LR(1)$-shift/reduce parser associated with $DCG$ is built as follows: The shift and goto entries come directly from the transitions of $DCG$, and for every state $s$, for every item
4.11. LR(1)-CHARACTERISTIC AUTOMATA

Figure 4.18: DFA for $C_{G_3}$

$(A \rightarrow \gamma, b)$ in $s$, enter an entry $rn$ for state $s$ and input $b$, where $A \rightarrow \gamma$ is production number $n$. If the resulting parser has no conflicts, we say that the grammar is an LR(1) grammar. The LR(1)-shift/reduce parser for $G_3$ is shown below:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$b$</td>
<td>$s2$</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>$r2$</td>
<td>$s3$</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>$r2$</td>
<td>$s3$</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>$r5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$r1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$s7$</td>
<td>$r1$</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>$r1$</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>acc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observe that there are three pairs of states, $(2, 3)$, $(4, 6)$, and $(5, 7)$, where both states in a common pair only differ by the lookahead symbols. We can merge the states corresponding to each pair, because the marked items are the same, but now, we have to allow lookahead sets. Thus, the merging of $(2, 3)$ yields

$$2': E \rightarrow a.Eb, \{b, \}$
$$E \rightarrow .aEb, \{b\}
$$E \rightarrow ., \{b\},$$

the merging of $(4, 6)$ yields

$$3': E \rightarrow aE.b, \{b, \}$

the merging of $(5, 7)$ yields

$$4': E \rightarrow aEb., \{b, \}.$$
We obtain a merged DFA with only five states, and the corresponding shift/reduce parser is given below:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s2'</td>
<td>r2</td>
<td>2'</td>
<td>8</td>
</tr>
<tr>
<td>2'</td>
<td>s2'</td>
<td>r2</td>
<td>3'</td>
<td>3'</td>
</tr>
<tr>
<td>3'</td>
<td>s4'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4'</td>
<td>r1</td>
<td>r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
</tr>
</tbody>
</table>

The reader should verify that this is the \textit{LALR}(1)-parser. The reader should also check that the \textit{SLR}(1)-parser is given below:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s2</td>
<td>r2</td>
<td>r2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>s2</td>
<td>r2</td>
<td>r2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r1</td>
<td>r1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
</tr>
</tbody>
</table>

The difference between the two parsing tables is that the \textit{LALR}(1)-lookahead sets are sharper than the \textit{SLR}(1)-lookahead sets. This is because the computation of the \textit{LALR}(1)-lookahead sets uses a sharper version of FOLLOW sets. It can also be shown that if a grammar is \textit{LALR}(1), then the merging of states of an \textit{LR}(1)-parser always succeeds and yields the \textit{LALR}(1) parser. Of course, this is a very inefficient way of producing \textit{LALR}(1) parsers, and much better methods exist, such as the graph method described in these notes. However, there are cases where the merging fails. Sufficient conditions for successful merging have been investigated, but there is still room for research in this area.
Chapter 5

RAM Programs, Turing Machines, and the Partial Recursive Functions

See the scanned version of this chapter found in the web page for CIS511:

Chapter 6

Universal RAM Programs and Undecidability of the Halting Problem

6.1 Pairing Functions

Pairing functions are used to encode pairs of integers into single integers, or more generally, finite sequences of integers into single integers. We begin by exhibiting a bijective pairing function $J : \mathbb{N}^2 \rightarrow \mathbb{N}$. The function $J$ has the graph partially showed below:

\[
\begin{array}{ccccccc}
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
& & & & & & \\
\end{array}
\]

The function $J$ corresponds to a certain way of enumerating pairs of integers. Note that the value of $x + y$ is constant along each diagonal, and consequently, we have

\[
J(x, y) = 1 + 2 + \cdots + (x + y) + x,
\]

\[
= ((x + y)(x + y + 1) + 2x)/2,
\]

\[
= ((x + y)^2 + 3x + y)/2,
\]

that is,

\[
J(x, y) = ((x + y)^2 + 3x + y)/2.
\]

Let $K : \mathbb{N} \rightarrow \mathbb{N}$ and $L : \mathbb{N} \rightarrow \mathbb{N}$ be the projection functions onto the axes, that is, the unique functions such that

\[
K(J(a, b)) = a \quad \text{and} \quad L(J(a, b)) = b,
\]
for all $a, b \in \mathbb{N}$.

Clearly, $J$ is primitive recursive, since it is given by a polynomial. It is not hard to prove that $J$ is injective and surjective, and that it is strictly monotonic in each argument, which means that for all $x, x', y, y' \in \mathbb{N}$, if $x < x'$ then $J(x, y) < J(x', y)$, and if $y < y'$ then $J(x, y) < J(x, y')$.

The projection functions can be computed explicitly, although this is a bit tricky. We only need to observe that by monotonicity of $J$, $x \leq J(x, y)$ and $y \leq J(x, y)$, and thus,

$$K(z) = \min(x \leq z)(\exists y \leq z)[J(x, y) = z],$$

and

$$L(z) = \min(y \leq z)(\exists x \leq z)[J(x, y) = z].$$

Therefore, $K$ and $L$ are primitive recursive. It can be verified that $J(K(z), L(z)) = z$, for all $z \in \mathbb{N}$.

More explicit formulae can be given for $K$ and $L$. If we define

$$Q_1(z) = \left\lfloor \left\lfloor \sqrt{8z + 1} \right\rfloor + 1 \right\rfloor / 2 - 1,$$

$$Q_2(z) = 2z - (Q_1(z))^2,$$

then it can be shown that

$$K(z) = \frac{1}{2}(Q_2(z) - Q_1(z))$$

$$L(z) = Q_1(z) - \frac{1}{2}(Q_2(z) - Q_1(z)).$$

In the above formula, the function $m \mapsto \left\lfloor \sqrt{m} \right\rfloor$ yields the largest integer $s$ such that $s^2 \leq m$. It can be computed by a RAM program.

The pairing function $J(x, y)$ is also denoted as $\langle x, y \rangle$, and $K$ and $L$ are also denoted as $\Pi_1$ and $\Pi_2$.

By induction, we can define bijections between $\mathbb{N}^n$ and $\mathbb{N}$ for all $n \geq 1$. We let $\langle z \rangle_1 = z$, $\langle x_1, x_2 \rangle_2 = \langle x_1, x_2 \rangle$, and

$$\langle x_1, \ldots, x_n, x_{n+1} \rangle_{n+1} = \langle x_1, \ldots, x_{n-1}, \langle x_n, x_{n+1} \rangle \rangle_n.$$
6.1. PAIRING FUNCTIONS

We can define a uniform projection function \( \Pi \) with the following property:
if \( z = \langle x_1, \ldots, x_n \rangle \), with \( n \geq 2 \), then
\[
\Pi(i, n, z) = x_i
\]
for all \( i \), where \( 1 \leq i \leq n \). The idea is to view \( z \) as a \( n \)-tuple, and \( \Pi(i, n, z) \) as the \( i \)-th component of that \( n \)-tuple. The function \( \Pi \) is defined by cases as follows:
\[
\begin{align*}
\Pi(i, 0, z) &= 0, \quad &\text{for all } i \geq 0, \\
\Pi(i, 1, z) &= z, \quad &\text{for all } i \geq 0, \\
\Pi(i, 2, z) &= \Pi_1(z), \quad &\text{if } 0 \leq i \leq 1, \\
\Pi(i, 2, z) &= \Pi_2(z), \quad &\text{for all } i \geq 2,
\end{align*}
\]
and for all \( n \geq 2 \),
\[
\Pi(i, n + 1, z) = \begin{cases} 
\Pi(i, n, z) & \text{if } 0 \leq i < n, \\
\Pi_1(\Pi(n, n, z)) & \text{if } i = n, \\
\Pi_2(\Pi(n, n, z)) & \text{if } i > n.
\end{cases}
\]

By a previous exercise, this is a legitimate primitive recursive definition. Some basic properties of \( \Pi \) are given as exercises. In particular, the following properties are easily shown:

(a) \( \langle 0, \ldots, 0 \rangle_n = 0, \langle x, 0 \rangle = \langle x, 0, \ldots, 0 \rangle_n; \)
(b) \( \Pi(0, n, z) = \Pi(1, n, z) \) and \( \Pi(i, n, z) = \Pi(n, n, z) \), for all \( i \geq n \) and all \( n, z \in \mathbb{N} \);
(c) \( \langle \Pi(1, n, z), \ldots, \Pi(n, n, z) \rangle_n = z \), for all \( n \geq 1 \) and all \( z \in \mathbb{N} \);
(d) \( \Pi(i, n, z) \leq z \), for all \( i, n, z \in \mathbb{N} \);
(e) There is a primitive recursive function \( \text{Large} \), such that,
\[
\Pi(i, n + 1, \text{Large}(n + 1, z)) = z,
\]
for \( i, n, z \in \mathbb{N} \).

As a first application, we observe that we need only consider partial recursive functions of a single argument. Indeed, let \( \varphi: \mathbb{N}^n \to \mathbb{N} \) be a partial recursive function of \( n \geq 2 \) arguments. Let
\[
\overline{\varphi}(z) = \varphi(\Pi(1, n, z), \ldots, \Pi(n, n, z)),
\]
for all \( z \in \mathbb{N} \). Then, \( \overline{\varphi} \) is a partial recursive function of a single argument, and \( \varphi \) can be recovered from \( \overline{\varphi} \), since
\[
\varphi(x_1, \ldots, x_n) = \overline{\varphi}(\langle x_1, \ldots, x_n \rangle).
\]
Thus, using \( \langle -, - \rangle \) and \( \Pi \) as coding and decoding functions, we can restrict our attention to functions of a single argument.

Next, we show that there exist coding and decoding functions between \( \Sigma^* \) and \( \{a_1\}^* \), and that partial recursive functions over \( \Sigma^* \) can be recoded as partial recursive functions over \( \{a_1\}^* \). Since \( \{a_1\}^* \) is isomorphic to \( \mathbb{N} \), this shows that we can restrict our attention to functions defined over \( \mathbb{N} \).
6.2 Equivalence of Alphabets

Given an alphabet \( \Sigma = \{a_1, \ldots, a_k\} \), strings over \( \Sigma \) can be ordered by viewing strings as numbers in a number system where the digits are \( a_1, \ldots, a_k \). In this number system, which is almost the number system with base \( k \), the string \( a_1 \) corresponds to zero, and \( a_k \) to \( k - 1 \). Hence, we have a kind of shifted number system in base \( k \). For example, if \( \Sigma = \{a, b, c\} \), a listing of \( \Sigma^* \) in the ordering corresponding to the number system begins with

\[
a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc,
\]

\[
aaa, aab,aac,aba,abb,abc,...
\]

Clearly, there is an ordering function from \( \Sigma^* \) to \( \mathbb{N} \) which is a bijection. Indeed, if \( u = a_{i_1} \cdots a_{i_n} \), this function \( f: \Sigma^* \rightarrow \mathbb{N} \) is given by

\[
f(u) = i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n.
\]

Since we also want a decoding function, we define the coding function \( C_k: \Sigma^* \rightarrow \Sigma^* \) as follows:

\[
C_k(\epsilon) = \epsilon, \quad \text{and if } u = a_{i_1} \cdots a_{i_n}, \text{ then } C_k(u) = a_1^{i_1 k^{n-1} + i_2 k^{n-2} + \cdots + i_{n-1} k + i_n}.
\]

The function \( C_k \) is primitive recursive, because

\[
C_k(\epsilon) = \epsilon, \quad C_k(xa_i) = C_k(x)a_i^i.
\]

The inverse of \( C_k \) is a function \( D_k: \{a_1\}^* \rightarrow \Sigma^* \). However, primitive recursive functions are total, and we need to extend \( D_k \) to \( \Sigma^* \). This is easily done by letting

\[
D_k(x) = D_k(a_1^{\lvert x \rvert})
\]

for all \( x \in \Sigma^* \). It remains to define \( D_k \) by primitive recursion over \( \{a_1\}^* \). For this, we introduce three auxiliary functions \( p, q, r \), defined as follows. Let

\[
p(\epsilon) = \epsilon, \quad p(xa_i) = xa_i, \quad \text{if } i \neq k, \quad p(xa_k) = p(x).
\]

Note that \( p(x) \) is the result of deteting consecutive \( a_k \)'s in the tail of \( x \). Let

\[
q(\epsilon) = \epsilon, \quad q(xa_i) = q(x)a_1.
\]
Note that \( q(x) = a_1^{\lfloor x \rfloor} \). Finally, let
\[
\begin{align*}
r(\epsilon) &= a_1, \\
r(xa_i) &= xa_{i+1}, & \text{if } i \neq k, \\
r(xa_k) &= xa_k.
\end{align*}
\]
The function \( r \) is almost the successor function, for the ordering. Then, the trick is that \( D_k(xa_i) \) is the successor of \( D_k(x) \) in the ordering, and if
\[ D_k(x) = ya_ja_k^n \]
with \( j \neq k \), since the successor of \( ya_ja_k^n \) is \( ya_{j+1}a_k^n \), we can use \( r \). Thus, we have
\[
\begin{align*}
D_k(\epsilon) &= \epsilon, \\
D_k(xa_i) &= r(p(D_k(x)))q(D_k(x) - p(D_k(x))).
\end{align*}
\]
Then, both \( C_k \) and \( D_k \) are primitive recursive, and \( C_k \circ D_k = D_k \circ C_k = \text{id} \).

Let \( \varphi: \Sigma^* \to \Sigma^* \) be a partial function over \( \Sigma^* \), and let
\[
\varphi^+(x_1, \ldots, x_n) = C_k(\varphi(D_k(x_1), \ldots, D_k(x_n))).
\]
The function \( \varphi^+ \) is defined over \( \{a_1\}^* \). Also, for any partial function \( \psi \) over \( \{a_1\}^* \), let
\[
\psi^*(x_1, \ldots, x_n) = D_k(\psi(C_k(x_1), \ldots, C_k(x_n))).
\]
We claim that if \( \psi \) is a partial recursive function over \( \{a_1\}^* \), then \( \psi^* \) is partial recursive over \( \Sigma^* \), and that if \( \varphi \) is a partial recursive function over \( \Sigma^* \), then \( \varphi^+ \) is partial recursive over \( \{a_1\}^* \).

First, \( \psi \) can be extended to \( \Sigma^* \) by letting
\[
\psi(x) = \psi(a_1^{\lfloor x \rfloor})
\]
for all \( x \in \Sigma^* \), and so, if \( \psi \) is partial recursive, then so is \( \psi^* \) by composition. This seems equally obvious for \( \varphi \) and \( \varphi^+ \), but there is a difficulty. The problem is that \( \varphi^+ \) is defined as a composition of functions over \( \Sigma^* \). We have to show how \( \varphi^+ \) can be defined directly over \( \{a_1\}^* \) without using any additional alphabet symbols. This is done in Machtey and Young [2], see Section 2.2, Proposition 2.2.3.

Pairing functions can also be used to prove that certain functions are primitive recursive, even though their definition is not a legal primitive recursive definition. For example, consider the Fibonacci function defined as follows:
\[
\begin{align*}
f(0) &= 1, \\
f(1) &= 2, \\
f(n + 2) &= f(n + 1) + f(n),
\end{align*}
\]
for all \( n \in \mathbb{N} \). This is not a legal primitive recursive definition, since \( f(n + 2) \) depends both on \( f(n+1) \) and \( f(n) \). In a primitive recursive definition, \( g(y+1, \overline{x}) \) is only allowed to depend upon \( g(y, \overline{x}) \).
Definition 6.1. Given any function $f : \mathbb{N}^n \to \mathbb{N}$, the function $\overline{f} : \mathbb{N}^{n+1} \to \mathbb{N}$ defined such that
$$\overline{f}(y, x) = \langle f(0, x), \ldots, f(y, x) \rangle_{y+1}$$
is called the course-of-value function for $f$.

The following lemma holds.

Lemma 6.1. Given any function $f : \mathbb{N}^n \to \mathbb{N}$, if $f$ is primitive recursive, then so is $\overline{f}$.

Proof. First, it is necessary to define a function $\text{con}$ such that if $x = \langle x_1, \ldots, x_m \rangle$ and $y = \langle y_1, \ldots, y_n \rangle$, where $m, n \geq 1$, then
$$\text{con}(m, x, y) = \langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle.$$ This fact is left as an exercise. Now, if $f$ is primitive recursive, let
$$\overline{f}(0, x) = f(0, x),$$
$$\overline{f}(y + 1, x) = \text{con}(y + 1, \overline{f}(y, x), f(y + 1, x)),$$
showing that $\overline{f}$ is primitive recursive. Conversely, if $\overline{f}$ is primitive recursive, then
$$f(y, x) = \Pi(y + 1, y + 1, \overline{f}(y, x)),$$
and so, $f$ is primitive recursive. \hfill \Box

Remark: Why is it that
$$\overline{f}(y + 1, x) = \langle \overline{f}(y, x), f(y + 1, x) \rangle$$
does not work?

We define course-of-value recursion as follows.

Definition 6.2. Given any two functions $g : \mathbb{N}^n \to \mathbb{N}$ and $h : \mathbb{N}^{n+2} \to \mathbb{N}$, the function $\overline{f} : \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by course-of-value recursion from $g$ and $h$ if
$$f(0, x) = g(x),$$
$$f(y + 1, x) = h(y, \overline{f}(y, x), x).$$

The following lemma holds.

Lemma 6.2. If $\overline{f} : \mathbb{N}^{n+1} \to \mathbb{N}$ is defined by course-of-value recursion from $g$ and $h$ and $g, h$ are primitive recursive, then $f$ is primitive recursive.
Proof. We prove that \( \overline{f} \) is primitive recursive. Then, by Lemma 6.1, \( f \) is also primitive recursive. To prove that \( \overline{f} \) is primitive recursive, observe that

\[
\overline{f}(0, \bar{x}) = g(\bar{x}), \\
\overline{f}(y + 1, \bar{x}) = \text{con}(y + 1, \overline{f}(y, \bar{x}), h(y, \overline{f}(y, \bar{x}), \bar{x})).
\]

\( \square \)

When we use Lemma 6.2 to prove that a function is primitive recursive, we rarely bother to construct a formal course-of-value recursion. Instead, we simply indicate how the value of \( f(y + 1, \bar{x}) \) can be obtained in a primitive recursive manner from \( f(0, \bar{x}) \) through \( f(y, \bar{x}) \). Thus, an informal use of Lemma 6.2 shows that the Fibonacci function is primitive recursive. A rigorous proof of this fact is left as an exercise.

### 6.3 Coding of RAM Programs

In this Section, we present a specific encoding of RAM programs which allows us to treat programs as integers. Encoding programs as integers also allows us to have programs that take other programs as input, and we obtain a \textit{universal program}. Universal programs have the property that given two inputs, the first one being the code of a program and the second one an input data, the universal program simulates the actions of the encoded program on the input data. A coding scheme is also called an indexing or a Gödel numbering, in honor to Gödel, who invented this technique.

From results of the previous Chapter, without loss of generality, we can restrict out attention to RAM programs computing partial functions of one argument over \( \mathbb{N} \). Furthermore, we only need the following kinds of instructions, each instruction being coded as shown below. Since we are considering functions over the natural numbers, which corresponds to a one-letter alphabet, there is only one kind of instruction of the form \textit{add} and \textit{jmp} (and \textit{add} increments by 1 the contents of the specified register \( R_j \)).

\[
\begin{align*}
N_i \text{ add } R_j & \quad \text{code} = \langle 1, i, j, 0 \rangle \\
N_i \text{ tail } R_j & \quad \text{code} = \langle 2, i, j, 0 \rangle \\
N_i \text{ continue } & \quad \text{code} = \langle 3, i, 1, 0 \rangle \\
N_i \text{ jmp } R_j & \quad \text{code} = \langle 4, i, j, k \rangle \\
N_i \text{ jmp } N_k & \quad \text{code} = \langle 5, i, j, k \rangle
\end{align*}
\]

Recall that a conditional jump causes a jump to the closest address \( N_k \) above or below iff \( R_j \) is nonzero, and if \( R_j \) is null, the next instruction is executed. We assume that all lines in a RAM program are numbered. This is always feasible, by labeling unnamed instructions with a new and unused line number.
The code of an instruction $I$ is denoted as $\#I$. To simplify the notation, we introduce the following decoding primitive recursive functions Typ, Nam, Reg, and Jmp, defined as follows:

\[
\begin{align*}
\text{Typ}(x) &= \Pi(1, 4, x), \\
\text{Nam}(x) &= \Pi(2, 4, x), \\
\text{Reg}(x) &= \Pi(3, 4, x), \\
\text{Jmp}(x) &= \Pi(4, 4, x).
\end{align*}
\]

The functions yield the type, line number, register name, and line number jumped to, if any, for an instruction coded by $x$. Note that we have no need to interpret the values of these functions if $x$ does not code an instruction.

We can define the primitive recursive predicate INST, such that $\text{INST}(x)$ holds iff $x$ codes an instruction. First, we need the connective $\supset$ (implies), defined such that $P \supset Q$ iff $\neg P \lor Q$.

Then, $\text{INST}(x)$ holds iff:

\[
\begin{align*}
[1 \leq \text{Typ}(x) \leq 5] \land [1 \leq \text{Reg}(x)] \land \\
[\text{Typ}(x) \leq 3 \supset \text{Jmp}(x) = 0] \land \\
[\text{Typ}(x) = 3 \supset \text{Reg}(x) = 1].
\end{align*}
\]

Program are coded as follows. If $P$ is a RAM program composed of the $n$ instructions $I_1, \ldots, I_n$, the code of $P$, denoted as $\#P$, is

\[
\#P = \langle n, \#I_1, \ldots, \#I_n \rangle.
\]

Recall from a previous exercise that

\[
\langle n, \#I_1, \ldots, \#I_n \rangle = \langle n, \langle \#I_1, \ldots, \#I_n \rangle \rangle.
\]

Also recall that

\[
\langle x, y \rangle = ((x + y)^2 + 3x + y)/2.
\]

Consider the following program Padd2 computing the function $\text{add2}: \mathbb{N} \rightarrow \mathbb{N}$ given by $\text{add2}(n) = n + 2$.

\[
\begin{align*}
I_1 : & \quad 1 \quad \text{add} \quad R1 \\
I_2 : & \quad 2 \quad \text{add} \quad R1 \\
I_3 : & \quad 3 \quad \text{continue}
\end{align*}
\]
We have
\[\#I_1 = \langle 1, 1, 1, 0 \rangle_4 = \langle 1, \langle 1, \langle 1, 0 \rangle \rangle \rangle = 37\]
\[\#I_2 = \langle 1, 2, 1, 0 \rangle_4 = \langle 1, \langle 2, \langle 1, 0 \rangle \rangle \rangle = 92\]
\[\#I_3 = \langle 3, 3, 1, 0 \rangle_4 = \langle 3, \langle 3, \langle 1, 0 \rangle \rangle \rangle = 234\]

and
\[\#Padd_2 = \langle 3, \#I_1, \#I_2, \#I_3 \rangle_4 = \langle 3, \langle 37, \langle 92, 234 \rangle \rangle \rangle = 1018748519973070618.\]

The codes get big fast!

We define the primitive recursive functions \(L_n, P_g,\) and \(L_i(x)\), such that:
\[
L_n(x) = \Pi(1, 2, x), \\
P_g(x) = \Pi(2, 2, x), \\
L_i(x) = \Pi(i, L_n(x), P_g(x)).
\]

The function \(L_n\) yields the length of the program (the number of instructions), \(P_g\) yields
the sequence of instructions in the program (really, a code for the sequence), and \(L_i(x)\)
yields the code of the \(i\)th instruction in the program. Again, if \(x\) does not code a program,
there is no need to interpret these functions. However, note that by a previous exercise, it
happens that
\[
L_0(x) = L_1(x), \quad \text{and} \\
L_i(L_n(x), x) = L_i(x), \quad \text{for all } i \geq x.
\]

The primitive recursive predicate \(\text{PROG}\) is defined such that \(\text{PROG}(x)\) holds iff \(x\) codes
a program. Thus, \(\text{PROG}(x)\) holds if each line codes an instruction, each jump has an
instruction to jump to, and the last instruction is a \text{continue}. Thus, \(\text{PROG}(x)\) holds iff
\[
\forall i \leq L_n(x)[i \geq 1 \supset \\
\text{INST}(L_i(x)) \land \text{Typ}(L_i(L_n(x), x)) = 3 \\
\land [\text{Typ}(L_i(x)) = 4 \supset \\
\exists j \leq i - 1[j \geq 1 \land \text{Nam}(L_j(x)) = \text{Jmp}(L_i(x))]]\land \\
[\text{Typ}(L_i(x)) = 5 \supset \\
\exists j \leq L_n(x)[j > i \land \text{Nam}(L_j(x)) = \text{Jmp}(L_i(x))]]]
\]

Note that we have used the fact proved as an exercise that if \(f\) is a primitive recursive
function and \(P\) is a primitive recursive predicate, then \(\exists x \leq f(y)P(x)\) is primitive recursive.

We are now ready to prove a fundamental result in the theory of algorithms. This result
points out some of the limitations of the notion of algorithm.
**Theorem 6.3. (Undecidability of the halting problem)** There is no RAM program \textbf{Decider} which halts for all inputs and has the following property when started with input \( x \) in register \( R1 \) and with input \( i \) in register \( R2 \) (the other registers being set to zero):

1. \textbf{Decider} halts with output 1 iff \( i \) codes a program that eventually halts when started on input \( x \) (all other registers set to zero).

2. \textbf{Decider} halts with output 0 in \( R1 \) iff \( i \) codes a program that runs forever when started on input \( x \) in \( R1 \) (all other registers set to zero).

3. If \( i \) does not code a program, then \textbf{Decider} halts with output 2 in \( R1 \).

**Proof.** Assume that \textbf{Decider} is such a RAM program, and let \( Q \) be the following program with a single input:

\[
\begin{align*}
\text{Program } Q \ (\text{code } q) \quad & \begin{cases} 
R2 & \leftarrow \text{R1} \\
R1 & \text{jmp} \quad N1a \\
R1 & \text{continue} \quad N1 \\
P & \text{continue}
\end{cases} \\
\text{sequent: } R2 & \leftarrow \text{R1}
\end{align*}
\]

Let \( i \) be the code of some program \( P \). The key point is that the termination behavior of \( Q \) on input \( i \) is exactly the opposite of the termination behavior of \textbf{Decider} on input \( i \) and code \( i \).

1. If \textbf{Decider} says that program \( P \) coded by \( i \) halts on input \( i \), then \( R1 \) just after the \texttt{continue} in line \( N1 \) contains 1, and \( Q \) loops forever.

2. If \textbf{Decider} says that program \( P \) coded by \( i \) loops forever on input \( i \), then \( R1 \) just after \texttt{continue} in line \( N1 \) contains 0, and \( Q \) halts.

The program \( Q \) can be translated into a program using only instructions of type 1, 2, 3, 4, 5, described previously, and let \( q \) be the code of the program \( Q \).

Let us see what happens if we run the program \( Q \) on input \( q \) in \( R1 \) (all other registers set to zero).

Just after execution of the assignment \( R2 \leftarrow R1 \), the program \textbf{Decider} is started with \( q \) in both \( R1 \) and \( R2 \). Since \textbf{Decider} is supposed to halt for all inputs, it eventually halts with output 0 or 1 in \( R1 \). If \textbf{Decider} halts with output 1 in \( R1 \), then \( Q \) goes into an infinite loop, while if \textbf{Decider} halts with output 0 in \( R1 \), then \( Q \) halts. But then, because of the definition of \textbf{Decider}, we see that \textbf{Decider} says that \( Q \) halts when started on input \( q \) iff \( Q \) loops forever on input \( q \), and that \( Q \) loops forever on input \( q \) iff \( Q \) halts on input \( q \), a contradiction. Therefore, \textbf{Decider} cannot exist. \( \Box \)
If we identify the notion of algorithm with that of a RAM program which halts for all inputs, the above theorem says that there is no algorithm for deciding whether a RAM program eventually halts for a given input. We say that the halting problem for RAM programs is undecidable (or unsolvable).

The above theorem also implies that the halting problem for Turing machines is undecidable. Indeed, if we had an algorithm for solving the halting problem for Turing machines, we could solve the halting problem for RAM programs as follows: first, apply the algorithm for translating a RAM program into an equivalent Turing machine, and then apply the algorithm solving the halting problem for Turing machines.

The argument is typical in computability theory and is called a “reducibility argument.”

Our next goal is to define a primitive recursive function that describes the computation of RAM programs. Assume that we have a RAM program \( P \) using \( n \) registers \( R_1, \ldots, R_n \), whose contents are denoted as \( r_1, \ldots, r_n \). We can code \( r_1, \ldots, r_n \) into a single integer \( \langle r_1, \ldots, r_n \rangle \). Conversely, every integer \( x \) can be viewed as coding the contents of \( R_1, \ldots, R_n \), by taking the sequence \( \Pi(1, n, x), \ldots, \Pi(n, n, x) \).

Actually, it is not necessary to know \( n \), the number of registers, if we make the following observation:

\[
\text{Reg}(\text{Line}(i, x)) \leq \text{Line}(i, x) \leq \text{Pg}(x)
\]

for all \( i, x \in \mathbb{N} \). Then, if \( x \) codes a program, then \( R_1, \ldots, R_x \) certainly include all the registers in the program. Also note that from a previous exercise,

\[
\langle r_1, \ldots, r_n, 0, \ldots, 0 \rangle = \langle r_1, \ldots, r_n, 0 \rangle.
\]

We now define the primitive recursive functions \( \text{Nextline} \), \( \text{Nextcont} \), and \( \text{Comp} \), describing the computation of RAM programs.

**Definition 6.3.** Let \( x \) code a program and let \( i \) be such that \( 1 \leq i \leq \text{Ln}(x) \). The following functions are defined:

1. \( \text{Nextline}(i, x, y) \) is the number of the next instruction to be executed after executing the \( i \)th instruction in the program coded by \( x \), where the contents of the registers is coded by \( y \).

2. \( \text{Nextcont}(i, x, y) \) is the code of the contents of the registers after executing the \( i \)th instruction in the program coded by \( x \), where the contents of the registers is coded by \( y \).

3. \( \text{Comp}(x, y, m) = \langle i, z \rangle \), where \( i \) and \( z \) are defined such that after running the program coded by \( x \) for \( m \) steps, where the initial contents of the program registers are coded by \( y \), the next instruction to be executed is the \( i \)th one, and \( z \) is the code of the current contents of the registers.

**Lemma 6.4.** The functions \( \text{Nextline} \), \( \text{Nextcont} \), and \( \text{Comp} \), are primitive recursive.
Proof. (1) Nextline\((i, x, y) = i + 1\), unless the \(i\)th instruction is a jump and the contents of the register being tested is nonzero:

\[
\text{Nextline}(i, x, y) = \\
\begin{cases} 
\max j \leq \text{Ln}(x)[j < i \land \text{Nam}(	ext{Line}(j, x)) = \text{Jmp}(	ext{Line}(i, x))] & \text{if Typ(Line}(i, x)) = 4 \land \Pi(\text{Reg(Line}(i, x)), x, y) \neq 0 \\
\min j \leq \text{Ln}(x)[j > i \land \text{Nam}(	ext{Line}(j, x)) = \text{Jmp}(	ext{Line}(i, x))] & \text{if Typ(Line}(i, x)) = 5 \land \Pi(\text{Reg(Line}(i, x)), x, y) \neq 0 \\
i + 1 & \text{otherwise.}
\end{cases}
\]

Note that according to this definition, if the \(i\)th line is the final \texttt{continue}, then Nextline signals that the program has halted by yielding

\[
\text{Nextline}(i, x, y) > \text{Ln}(x).
\]

(2) We need two auxiliary functions \text{Add} and \text{Sub} defined as follows.

\text{Add}(j, x, y) is the number coding the contents of the registers used by the program coded by \(x\) after register \(R_j\) coded by \(\Pi(j, x, y)\) has been increased by 1, and

\text{Sub}(j, x, y) codes the contents of the registers after register \(R_j\) has been decremented by 1 (\(y\) codes the previous contents of the registers). It is easy to see that

\[
\text{Sub}(j, x, y) = \min z \leq y[\Pi(j, x, z) = \Pi(j, x, y) - 1 \\
\land \forall k \leq x[0 < k \neq j \supset \Pi(k, x, z) = \Pi(k, x, y)]].
\]

The definition of \text{Add} is slightly more tricky. We leave as an exercise to the reader to prove that:

\[
\text{Add}(j, x, y) = \min z \leq \text{Large}(x, y + 1) \\
[\Pi(j, x, z) = \Pi(j, x, y) + 1 \land \forall k \leq x[0 < k \neq j \supset \Pi(k, x, z) = \Pi(k, x, y)]],
\]

where the function \text{Large} is the function defined in an earlier exercise. Then

\[
\text{Nextcont}(i, x, y) = \\
\begin{cases} 
\text{Add(Reg(Line}(i, x), x, y) & \text{if Typ(Line}(i, x)) = 1 \\
\text{Sub(Reg(Line}(i, x), x, y) & \text{if Typ(Line}(i, x)) = 2 \\
y & \text{if Typ(Line}(i, x)) \geq 3.
\end{cases}
\]

(3) Recall that \(\Pi_1(z) = \Pi(1, 2, z)\) and \(\Pi_2(z) = \Pi(2, 2, z)\). The function \text{Comp} is defined by primitive recursion as follows:

\[
\text{Comp}(x, y, 0) = \langle 1, y \rangle \\
\text{Comp}(x, y, m + 1) = \langle \text{Nextline}(\Pi_1(\text{Comp}(x, y, m)), x, \Pi_2(\text{Comp}(x, y, m))), \\
\text{Nextcont}(\Pi_1(\text{Comp}(x, y, m)), x, \Pi_2(\text{Comp}(x, y, m))) \rangle.
\]
Recall that $\Pi_1(\text{Comp}(x, y, m))$ is the number of the next instruction to be executed and that $\Pi_2(\text{Comp}(x, y, m))$ codes the current contents of the registers.

We can now reprove that every RAM computable function is partial recursive. Indeed, assume that $x$ codes a program $P$.

We define the partial function $\text{End}$ so that for all $x, y$, where $x$ codes a program and $y$ codes the contents of its registers, $\text{End}(x, y)$ is the number of steps for which the computation runs before halting, if it halts. If the program does not halt, then $\text{End}(x, y)$ is undefined. Since

$$\text{End}(x, y) = \min m[\Pi_1(\text{Comp}(x, y, m)) = \text{Ln}(x)],$$

If $y$ is the value of the register $R1$ before the program $P$ coded by $x$ is started, recall that the contents of the registers is coded by $\langle y, 0 \rangle$. Noticing that 0 and 1 do not code programs, we note that if $x$ codes a program, then $x \geq 2$, and $\Pi_1(z) = \Pi(1, x, z)$ is the contents of $R1$ as coded by $z$.

Since $\text{Comp}(x, y, m) = \langle i, z \rangle$, we have

$$\Pi_1(\text{Comp}(x, y, m)) = i,$$

where $i$ is the number (index) of the instruction reached after running the program $P$ coded by $x$ with initial values of the registers coded by $y$ for $m$ steps. Thus, $P$ halts if $i$ is the last instruction in $P$, namely $\text{Ln}(x)$, iff

$$\Pi_1(\text{Comp}(x, y, m)) = \text{Ln}(x).$$

$\text{End}$ is a partial recursive function; it can be computed by a RAM program involving only one while loop searching for the number of steps $m$. However, in general, $\text{End}$ is not a total function.

If $\varphi$ is the partial recursive function computed by the program $P$ coded by $x$, then we have

$$\varphi(y) = \Pi_1(\Pi_2(\text{Comp}(x, \langle y, 0 \rangle, \text{End}(x, \langle y, 0 \rangle)))).$$

This is because if $m = \text{End}(x, \langle y, 0 \rangle)$ is the number of steps after which the program $P$ coded by $x$ halts on input $y$, then

$$\text{Comp}(x, \langle y, 0 \rangle, m)) = \langle \text{Ln}(x), z \rangle,$$

where $z$ is the code of the register contents when the program stops. Consequently

$$z = \Pi_2(\text{Comp}(x, \langle y, 0 \rangle, m))$$

$$z = \Pi_2(\text{Comp}(x, \langle y, 0 \rangle, \text{End}(x, \langle y, 0 \rangle))).$$

The value of the register $R1$ is $\Pi_1(z)$, that is

$$\varphi(y) = \Pi_1(\Pi_2(\text{Comp}(x, \langle y, 0 \rangle, \text{End}(x, \langle y, 0 \rangle)))).$$
Observe that \( \varphi \) is written in the form \( \varphi = g \circ \min f \), for some primitive recursive functions \( f \) and \( g \).

We can also exhibit a partial recursive function which enumerates all the unary partial recursive functions. It is a universal function.

Abusing the notation slightly, we will write \( \varphi(x, y) \) for \( \varphi(\langle x, y \rangle) \), viewing \( \varphi \) as a function of two arguments (however, \( \varphi \) is really a function of a single argument). We define the function \( \varphi_{univ} \) as follows:

\[
\varphi_{univ}(x, y) = \begin{cases} 
\Pi_1(\Pi_2(\text{Comp}(x, \langle y, 0 \rangle, \text{End}(x, \langle y, 0 \rangle)))) & \text{if PROG}(x), \\
\text{undefined} & \text{otherwise}.
\end{cases}
\]

The function \( \varphi_{univ} \) is a partial recursive function with the following property: for every \( x \) coding a RAM program \( P \), for every input \( y \),

\[
\varphi_{univ}(x, y) = \varphi_x(y),
\]

the value of the partial recursive function \( \varphi_x \) computed by the RAM program \( P \) coded by \( x \). If \( x \) does not code a program, then \( \varphi_{univ}(x, y) \) is undefined for all \( y \).

By Lemma 1.4.1 (in Chapter 4), \( \varphi_{univ} \) is not recursive. Indeed, being an enumerating function for the partial recursive functions, it is an enumerating function for the total recursive functions, and thus, it cannot be recursive. Being a partial function saves us from a contradiction.

The existence of the function \( \varphi_{univ} \) leads us to the notion of an indexing of the RAM programs.

We can define a listing of the RAM programs as follows. If \( x \) codes a program (that is, if \( \text{PROG}(x) \) holds) and \( P \) is the program that \( x \) codes, we call this program \( P \) the \( x \)th RAM program and denote it as \( P_x \). If \( x \) does not code a program, we let \( P_x \) be the program that diverges for every input:

\[
\begin{array}{c|c|c}
N1 & \text{add} & R1 \\
N1 & R1 & \text{jmp} \\
N1 & \text{continue} & N1a
\end{array}
\]

Therefore, in all cases, \( P_x \) stands for the \( x \)th RAM program. Thus, we have a listing of RAM programs, \( P_0, P_1, P_2, P_3, \ldots \), such that every RAM program (of the restricted type considered here) appears in the list exactly once, except for the “infinite loop” program. For example, the program Padd2 (adding 2 to an integer) appears as

\[
P_{1018748519973070618}.
\]

In particular, note that \( \varphi_{univ} \) being a partial recursive function, it is computed by some RAM program UNIV that has a code \( univ \) and is the program \( P_{univ} \) in the list.

Having an indexing of the RAM programs, we also have an indexing of the partial recursive functions.
Definition 6.4. For every integer \( x \geq 0 \), we let \( P_x \) be the RAM program coded by \( x \) as defined earlier, and \( \varphi_x \) be the partial recursive function computed by \( P_x \).

For example, the function \( \text{add2} \) (adding 2 to an integer) appears as
\[
\varphi_{1018748519973070618}.
\]

Remark: Kleene used the notation \( \{ x \} \) for the partial recursive function coded by \( x \). Due to the potential confusion with singleton sets, we follow Rogers, and use the notation \( \varphi_x \).

It is important to observe that different programs \( P_x \) and \( P_y \) may compute the same function, that is, while \( P_x \neq P_y \) for all \( x \neq y \), it is possible that \( \varphi_x = \varphi_y \). In fact, it is undecidable whether \( \varphi_x = \varphi_y \).

The existence of the universal function \( \varphi_{\text{univ}} \) is sufficiently important to be recorded in the following Lemma.

Lemma 6.5. For the indexing of RAM programs defined earlier, there is a universal partial recursive function \( \varphi_{\text{univ}} \) such that, for all \( x, y \in \mathbb{N} \), if \( \varphi_x \) is the partial recursive function computed by \( P_x \), then
\[
\varphi_x(y) = \varphi_{\text{univ}}(\langle x, y \rangle).
\]

The program UNIV computing \( \varphi_{\text{univ}} \) can be viewed as an interpreter for RAM programs. By giving the universal program UNIV the “program” \( x \) and the “data” \( y \), we get the result of executing program \( P_x \) on input \( y \). We can view the RAM model as a stored program computer.

By Theorem 6.3 and Lemma 6.5, the halting problem for the single program UNIV is undecidable. Otherwise, the halting problem for RAM programs would be decidable, a contradiction. It should be noted that the program UNIV can actually be written (with a certain amount of pain).

The object of the next Section is to show the existence of Kleene’s \( T \)-predicate. This will yield another important normal form. In addition, the \( T \)-predicate is a basic tool in recursion theory.

6.4  Kleene’s \( T \)-Predicate

In Section 6.3, we have encoded programs. The idea of this Section is to also encode computations of RAM programs. Assume that \( x \) codes a program, that \( y \) is some input (not a code), and that \( z \) codes a computation of \( P_x \) on input \( y \). The predicate \( T(x, y, z) \) is defined as follows:

\( T(x, y, z) \) holds iff \( x \) codes a RAM program, \( y \) is an input, and \( z \) codes a halting computation of \( P_x \) on input \( y \).
We will show that $T$ is primitive recursive. First, we need to encode computations. We say that $z$ codes a computation of length $n \geq 1$ if
\[ z = \langle n + 2, \langle 1, y_0 \rangle, \langle i_1, y_1 \rangle, \ldots, \langle i_n, y_n \rangle \rangle, \]
where each $i_j$ is the physical location of the next instruction to be executed and each $y_j$ codes the contents of the registers just before execution of the instruction at the location $i_j$. Also, $y_0$ codes the initial contents of the registers, that is, $y_0 = \langle y, 0 \rangle$, for some input $y$. We let $L_n(z) = \Pi_1(z)$. Note that $i_j$ denotes the physical location of the next instruction to be executed in the sequence of instructions constituting the program coded by $x$, and not the line number (label) of this instruction. Thus, the first instruction to be executed is in location 1, $1 \leq i_j \leq L_n(x)$, and $i_{n-1} = L_n(x)$. Since the last instruction which is executed is the last physical instruction in the program, namely, a `continue`, there is no next instruction to be executed after that, and $i_n$ is irrelevant. Writing the definition of $T$ is a little simpler if we let $i_n = L_n(x) + 1$.

**Definition 6.5.** The $T$-predicate is the primitive recursive predicate defined as follows:

\[ T(x, y, z) \text{ iff } \text{PROG}(x) \text{ and } (L_n(z) \geq 3) \text{ and } \forall j \leq L_n(z) - 3[0 \leq j \supset \text{Nextline}(\Pi_1(\Pi(j + 2, L_n(z), z)), x, \Pi_2(\Pi(j + 2, L_n(z), z))) = \Pi_1(\Pi(j + 3, L_n(z), z)) \text{ and } \text{Nextcont}(\Pi_1(\Pi(j + 2, L_n(z), z)), x, \Pi_2(\Pi(j + 2, L_n(z), z))) = \Pi_2(\Pi(j + 3, L_n(z), z)) \text{ and } \Pi_1(\Pi(L_n(z) - 1, L_n(z), z)) = L_n(x) \text{ and } \Pi_1(\Pi(2, L_n(z), z)) = 1 \text{ and } y = \Pi_1(\Pi_2(\Pi(2, L_n(z), z))) \text{ and } \Pi_2(\Pi_2(\Pi(2, L_n(z), z))) = 0] \]

The reader can verify that $T(x, y, z)$ holds iff $x$ codes a RAM program, $y$ is an input, and $z$ codes a halting computation of $P_x$ on input $y$. In order to extract the output of $P_x$ from $z$, we define the primitive recursive function $\text{Res}$ as follows:
\[ \text{Res}(z) = \Pi_1(\Pi_2(\Pi(L_n(z), L_n(z), z))). \]

The explanation for this formula is that $\text{Res}(z)$ are the contents of register $R1$ when $P_x$ halts, that is, $\Pi_1(y_{L_n(z)})$. Using the $T$-predicate, we get the so-called Kleene normal form.

**Theorem 6.6. (Kleene Normal Form)** Using the indexing of the partial recursive functions defined earlier, we have
\[ \varphi_x(y) = \text{Res}[\min z(T(x, y, z))], \]
where $T(x, y, z)$ and $\text{Res}$ are primitive recursive.

Note that the universal function $\varphi_{\text{univ}}$ can be defined as
\[ \varphi_{\text{univ}}(x, y) = \text{Res}[\min z(T(x, y, z))]. \]
There is another important property of the partial recursive functions, namely, that composition is effective. We need two auxiliary primitive recursive functions. The function Conprogs creates the code of the program obtained by concatenating the programs \( P_x \) and \( P_y \), and for \( i \geq 2 \), Cumclr\((i)\) is the code of the program which clears registers \( R_2, \ldots, R_i \). To get Cumclr, we can use the function clr\((i)\) such that clr\((i)\) is the code of the program

\[
\begin{align*}
N1 & \text{ tail } Ri \\
N1 & Ri \text{ jmp } N1a \\
N & \text{ continue }
\end{align*}
\]

We leave it as an exercise to prove that clr, Conprogs, and Cumclr, are primitive recursive.

**Theorem 6.7.** There is a primitive recursive function \( c \) such that

\[
\varphi_{c(x,y)} = \varphi_x \circ \varphi_y.
\]

**Proof.** If both \( x \) and \( y \) code programs, then \( \varphi_x \circ \varphi_y \) can be computed as follows: Run \( P_y \), clear all registers but \( R_1 \), then run \( P_x \). Otherwise, let loop be the index of the infinite loop program:

\[
c(x, y) = \begin{cases} 
\text{Conprogs}(y, \text{Conprogs}(\text{Cumclr}(y), x)) & \text{if PROG}(x) \text{ and PROG}(y) \\
\text{loop} & \text{otherwise.}
\end{cases}
\]

\( \square \)
Chapter 7

Elementary Recursive Function Theory

7.1 Acceptable Indexings

In Chapter 6, we have exhibited a specific indexing of the partial recursive functions by encoding the RAM programs. Using this indexing, we showed the existence of a universal function $\varphi_{\text{univ}}$ and of a recursive function $c$, with the property that for all $x, y \in \mathbb{N}$,

$$\varphi_c(x, y) = \varphi_x \circ \varphi_y.$$  

It is natural to wonder whether the same results hold if a different coding scheme is used or if a different model of computation is used, for example, Turing machines. In other words, we would like to know if our results depend on a specific coding scheme or not.

Our previous results showing the characterization of the partial recursive functions being independent of the specific model used, suggests that it might be possible to pinpoint certain properties of coding schemes which would allow an axiomatic development of recursive function theory. What we are aiming at is to find some simple properties of “nice” coding schemes that allow one to proceed without using explicit coding schemes, as long as the above properties hold.

Remarkably, such properties exist. Furthermore, any two coding schemes having these properties are equivalent in a strong sense (effectively equivalent), and so, one can pick any such coding scheme without any risk of losing anything else because the wrong coding scheme was chosen. Such coding schemes, also called indexings, or G"odel numberings, or even programming systems, are called acceptable indexings.

**Definition 7.1.** An indexing of the partial recursive functions is an infinite sequence $\varphi_0, \varphi_1, \ldots$, of partial recursive functions that includes all the partial recursive functions of one argument (there might be repetitions, this is why we are not using the term enumeration). An indexing is universal if it contains the partial recursive function $\varphi_{\text{univ}}$ such that

$$\varphi_{\text{univ}}(i, x) = \varphi_i(x)$$
for all $i, x \in \mathbb{N}$. An indexing is acceptable if it is universal and if there is a total recursive function $c$ for composition, such that

$$\varphi_{c(i, j)} = \varphi_i \circ \varphi_j$$

for all $i, j \in \mathbb{N}$.

From Chapter 6, we know that the specific indexing of the partial recursive functions given for RAM programs is acceptable. Another characterization of acceptable indexings left as an exercise is the following: an indexing $\psi_0, \psi_1, \psi_2, \ldots$ of the partial recursive functions is acceptable iff there exists a total recursive function $f$ translating the RAM indexing of Section 6.3 into the indexing $\psi_0, \psi_1, \psi_2, \ldots$, that is,

$$\varphi_i = \psi_{f(i)}$$

for all $i \in \mathbb{N}$.

A very useful property of acceptable indexings is the so-called “s-m-n Theorem”. Using the slightly loose notation $\varphi(x_1, \ldots, x_n)$ for $\varphi((x_1, \ldots, x_n))$, the s-m-n theorem says the following. Given a function $\varphi$ considered as having $m + n$ arguments, if we fix the values of the first $m$ arguments and we let the other $n$ arguments vary, we obtain a function $\psi$ of $n$ arguments. Then, the index of $\psi$ depends in a recursive fashion upon the index of $\varphi$ and the first $m$ arguments $x_1, \ldots, x_m$. We can “pull” the first $m$ arguments of $\varphi$ into the index of $\psi$.

**Theorem 7.1.** (The “s-m-n Theorem”) For any acceptable indexing $\varphi_0, \varphi_1, \ldots$, there is a total recursive function $s$, such that, for all $i, m, n \geq 1$, for all $x_1, \ldots, x_m$ and all $y_1, \ldots, y_n$, we have

$$\varphi_{s(i,m,x_1,\ldots,x_m)}(y_1, \ldots, y_n) = \varphi_i(x_1, \ldots, x_m, y_1, \ldots, y_n).$$

**Proof.** First, note that the above identity is really

$$\varphi_{s(i,m,(x_1,\ldots,x_m))}(y_1, \ldots, y_n) = \varphi_i((x_1, \ldots, x_m, y_1, \ldots, y_n)).$$

Recall that there is a primitive recursive function $\text{Con}$ such that

$$\text{Con}(m, (x_1, \ldots, x_m), (y_1, \ldots, y_n)) = (x_1, \ldots, x_m, y_1, \ldots, y_n)$$

for all $x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{N}$. Hence, a recursive function $s$ such that

$$\varphi_{s(i,m,x)}(y) = \varphi_i(\text{Con}(m, x, y))$$

will do. We define some auxiliary primitive recursive functions as follows:

$$P(y) = \langle 0, y \rangle \quad \text{and} \quad Q(\langle x, y \rangle) = \langle x + 1, y \rangle.$$
Since we have an indexing of the partial recursive functions, there are indices $p$ and $q$ such that $P = \varphi_p$ and $Q = \varphi_q$. Let $R$ be defined such that

$$R(0) = p,$$
$$R(x + 1) = c(q, R(x)),$$

where $c$ is the recursive function for composition given by the indexing. We leave as an exercise to prove that

$$\varphi_{R(x)}(y) = \langle x, y \rangle$$

for all $x, y \in \mathbb{N}$. Also, recall that $\langle x, y, z \rangle = \langle x, \langle y, z \rangle \rangle$, by definition of pairing. Then, we have

$$\varphi_{R(x)} \circ \varphi_{R(y)}(z) = \varphi_{R(x)}(\langle y, z \rangle) = \langle x, y, z \rangle.$$

Finally, let $k$ be an index for the function $\text{Con}$, that is, let

$$\varphi_k(\langle m, x, y \rangle) = \text{Con}(m, x, y).$$

Define $s$ by

$$s(i, m, x) = c(i, c(k, c(R(m), R(x)))).$$

Then, we have

$$\varphi_{s(i, m, x)}(y) = \varphi_i \circ \varphi_k \circ \varphi_{R(m)} \circ \varphi_{R(x)}(y) = \varphi_i(\text{Con}(m, x, y)),$$

as desired. Notice that if the composition function $c$ is primitive recursive, then $s$ is also primitive recursive. In particular, for the specific indexing of the RAM programs given in Section 6.3, the function $s$ is primitive recursive.

As a first application of the s-m-n Theorem, we show that any two acceptable indexings are effectively inter-translatable.

**Theorem 7.2.** Let $\varphi_0, \varphi_1, \ldots$, be a universal indexing, and let $\psi_0, \psi_1, \ldots$, be any indexing with a total recursive $s$-1-1 function, that is, a function $s$ such that

$$\psi_{s(i, x)}(y) = \psi_i(x, y)$$

for all $i, x, y \in \mathbb{N}$. Then, there is a total recursive function $t$ such that $\varphi_i = \psi_{t(i)}$.

**Proof.** Let $\varphi_{\text{univ}}$ be a universal partial recursive function for the indexing $\varphi_0, \varphi_1, \ldots$. Since $\psi_0, \psi_1, \ldots$, is also an indexing, $\varphi_{\text{univ}}$ occurs somewhere in the second list, and thus, there is some $k$ such that $\varphi_{\text{univ}} = \varphi_k$. Then, we have

$$\psi_{s(k, i)}(x) = \psi_k(i, x) = \varphi_{\text{univ}}(i, x) = \varphi_i(x),$$

for all $i, x \in \mathbb{N}$. Therefore, we can take the function $t$ to be the function defined such that

$$t(i) = s(k, 1, i)$$

for all $i \in \mathbb{N}$. \qed
Using Theorem 7.2, if we have two acceptable indexings \( \varphi_0, \varphi_1, \ldots \), and \( \psi_0, \psi_1, \ldots \), there exist total recursive functions \( t \) and \( u \) such that

\[
\varphi_i = \psi_t(i) \quad \text{and} \quad \psi_i = \varphi_u(i)
\]

for all \( i \in \mathbb{N} \). Also note that if the composition function \( c \) is primitive recursive, then any s-m-n function is primitive recursive, and the translation functions are primitive recursive. Actually, a stronger result can be shown. It can be shown that for any two acceptable indexings, there exist total recursive \textit{injective} and \textit{surjective} translation functions. In other words, any two acceptable indexings are recursively isomorphic (Roger’s isomorphism theorem). Next, we turn to algorithmically unsolvable, or \textit{undecidable}, problems.

### 7.2 Undecidable Problems

We saw in Section 6.3 that the halting problem for RAM programs is undecidable. In this section, we take a slightly more general approach to study the undecidability of problems, and give some tools for resolving decidability questions.

First, we prove again the undecidability of the halting problem, but this time, for \textit{any} indexing of the partial recursive functions.

**Theorem 7.3. (Halting Problem, Abstract Version)** Let \( \psi_0, \psi_1, \ldots \), be any indexing of the partial recursive functions. Then, the function \( f \) defined such that

\[
f(x, y) = \begin{cases} 
1 & \text{if } \psi_x(y) \text{ is defined}, \\
0 & \text{if } \psi_x(y) \text{ is undefined},
\end{cases}
\]

is not recursive.

**Proof.** Assume that \( f \) is recursive, and let \( g \) be the function defined such that

\[
g(x) = f(x, x)
\]

for all \( x \in \mathbb{N} \). Then \( g \) is also recursive. Let \( \theta \) be the function defined such that

\[
\theta(x) = \begin{cases} 
0 & \text{if } g(x) = 0, \\
\text{undefined} & \text{if } g(x) = 1.
\end{cases}
\]

We claim that \( \theta \) is not even partial recursive. Observe that \( \theta \) is such that

\[
\theta(x) = \begin{cases} 
0 & \text{if } \psi_x(x) \text{ is undefined}, \\
\text{undefined} & \text{if } \psi_x(x) \text{ is defined}.
\end{cases}
\]

If \( \theta \) was partial recursive, it would occur in the list as some \( \psi_i \), and we would have

\[
\theta(i) = \psi_i(i) = 0 \quad \text{iff} \quad \psi_i(i) \text{ is undefined},
\]

a contradiction. Therefore, \( f \) and \( g \) can’t be recursive. \( \square \)
Observe that the proof of Theorem 7.3 does not use the fact that the indexing is universal or acceptable, and thus, the theorem holds for any indexing of the partial recursive functions. The function $g$ defined in the proof of Theorem 7.3 is the characteristic function of a set denoted as $K$, where

$$K = \{ x \mid \psi_x(x) \text{ is defined} \}.$$ 

Given any set, $X$, for any subset, $A \subseteq X$, of $X$, recall that the characteristic function, $C_A$ (or $\chi_A$), of $A$ is the function, $C_A : X \to \{0, 1\}$, defined so that, for all $x \in X$,

$$C_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A. \end{cases}$$

The set $K$ is an example of a set which is not recursive. Since this fact is quite important, we give the following definition:

**Definition 7.2.** A subset, $A$, of $\Sigma^*$ (or a subset, $A$, of $\mathbb{N}$) is recursive (or decidable) iff its characteristic function, $C_A$, is a total recursive function.

Using Definition 7.2, Theorem 7.3 can be restated as follows.

**Lemma 7.4.** For any indexing $\varphi_0, \varphi_1, \ldots$ of the partial recursive functions (over $\Sigma^*$ or $\mathbb{N}$), the set $K = \{ x \mid \varphi_x(x) \text{ is defined} \}$ is not recursive.

Recursive sets allow us to define the concept of a decidable (or undecidable) problem. The idea is to generalize the situation described in Section 6.3 and Section 6.4, where a set of objects, the RAM programs, is encoded into a set of natural numbers, using a coding scheme.

**Definition 7.3.** Let $C$ be a countable set of objects, and let $P$ be a property of objects in $C$. We view $P$ as the set

$$\{ a \in C \mid P(a) \}.$$ 

A coding-scheme is an injective function $\# : C \to \mathbb{N}$ that assigns a unique code to each object in $C$. The property $P$ is decidable (relative to $\#$) iff the set $\{ \#(a) \mid a \in C \text{ and } P(a) \}$ is recursive. The property $P$ is undecidable (relative to $\#$) iff the set $\{ \#(a) \mid a \in C \text{ and } P(a) \}$ is not recursive.

Observe that the decidability of a property $P$ of objects in $C$ depends upon the coding scheme $\#$. Thus, if we are cheating in using a non-effective coding scheme, we may declare that a property is decidable even though it is not decidable in some reasonable coding scheme. Consequently, we require a coding scheme $\#$ to be effective in the following sense. Given any object $a \in C$, we can effectively (i.e., algorithmically) determine its code $\#(a)$. Conversely, given any integer $n \in \mathbb{N}$, we should be able to tell effectively if $n$ is the code of some object in $C$, and if so, to find this object. In practice, it is always possible to describe the objects
in $C$ as strings over some (possibly complex) alphabet $\Sigma$ (sets of trees, graphs, etc). In such cases, the coding schemes are recursive functions from $\Sigma^*$ to $\mathbb{N} = \{a_1\}^*$.

For example, let $C = \mathbb{N} \times \mathbb{N}$, where the property $P$ is the equality of the partial functions $\varphi_x$ and $\varphi_y$. We can use the pairing function $\langle -, - \rangle$ as a coding function, and the problem is formally encoded as the recursiveness of the set

$$\{ \langle x, y \rangle \mid x, y \in \mathbb{N}, \varphi_x = \varphi_y \}.$$ 

In most cases, we don’t even bother to describe the coding scheme explicitly, knowing that such a description is routine, although perhaps tedious.

We now show that most properties about programs (except the trivial ones) are undecidable. First, we show that it is undecidable whether a RAM program halts for every input. In other words, it is undecidable whether a procedure is an algorithm. We actually prove a more general fact.

**Lemma 7.5.** For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial recursive functions, the set

$$\text{TOTAL} = \{ x \mid \varphi_x \text{ is a total function} \}$$

is not recursive.

**Proof.** The proof uses a technique known as reducibility. We try to reduce a set $A$ known to be nonrecursive to TOTAL via a recursive function $f : A \to \text{TOTAL}$, so that

$$x \in A \iff f(x) \in \text{TOTAL}.$$ 

If TOTAL were recursive, its characteristic function $g$ would be recursive, and thus, the function $g \circ f$ would be recursive, a contradiction, since $A$ is assumed to be nonrecursive. In the present case, we pick $A = K$. To find the recursive function $f : K \to \text{TOTAL}$, we use the s-m-n Theorem. Let $\theta$ be the function defined below: for all $x, y \in \mathbb{N}$,

$$\theta(x, y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K. \end{cases}$$ 

Note that $\theta$ does not depend on $y$. The function $\theta$ is partial recursive. Indeed, we have

$$\theta(x, y) = \varphi_x(x) = \varphi_{\text{univ}}(x, x).$$ 

Thus, $\theta$ has some index $j$, so that $\theta = \varphi_j$, and by the s-m-n Theorem, we have

$$\varphi_{s(j, 1, x)}(y) = \varphi_j(x, y) = \theta(x, y).$$ 

Let $f$ be the recursive function defined such that

$$f(x) = s(j, 1, x)$$
for all $x \in \mathbb{N}$. Then, we have

$$\varphi_{f(x)}(y) = \begin{cases} \varphi_x(x) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K \end{cases}$$

for all $y \in \mathbb{N}$. Thus, observe that $\varphi_{f(x)}$ is a total function iff $x \in K$, that is,

$$x \in K \iff f(x) \in \text{TOTAL},$$

where $f$ is recursive. As we explained earlier, this shows that TOTAL is not recursive. 

The above argument can be generalized to yield a result known as Rice’s Theorem. Let $\varphi_0, \varphi_1, \ldots$ be any indexing of the partial recursive functions, and let $C$ be any set of partial recursive functions. We define the set $P_C$ as

$$P_C = \{ x \in \mathbb{N} \mid \varphi_x \in C \}.$$ 

We can view $C$ as a property of some of the partial recursive functions. For example

$$C = \{ \text{all total recursive functions} \}.$$ 

We say that $C$ is nontrivial if $C$ is neither empty nor the set of all partial recursive functions. Equivalently $C$ is nontrivial iff $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$. We may think of $P_C$ as the set of programs computing the functions in $C$.

**Theorem 7.6.** (Rice’s Theorem) For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial recursive functions, for any set $C$ of partial recursive functions, the set

$$P_C = \{ x \in \mathbb{N} \mid \varphi_x \in C \}$$

is nonrecursive unless $C$ is trivial.

**Proof.** Assume that $C$ is nontrivial. A set is recursive iff its complement is recursive (the proof is trivial). Hence, we may assume that the totally undefined function is not in $C$, and since $C \neq \emptyset$, let $\psi$ be some other function in $C$. We produce a recursive function $f$ such that

$$\varphi_{f(x)}(y) = \begin{cases} \psi(y) & \text{if } x \in K, \\ \text{undefined} & \text{if } x \notin K \end{cases}$$

for all $y \in \mathbb{N}$. We get $f$ by using the s-m-n Theorem. Let $\psi = \varphi_i$, and define $\theta$ as follows:

$$\theta(x, y) = \varphi_{\text{univ}}(i, y) + (\varphi_{\text{univ}}(x, x) - \varphi_{\text{univ}}(x, x)),$$

where $-$ is the primitive recursive function for truncated subtraction. Clearly, $\theta$ is partial recursive, and let $\theta = \varphi_j$. By the s-m-n Theorem, we have

$$\varphi_{s(j,1,x)}(y) = \varphi_j(x, y) = \theta(x, y)$$
for all $x, y \in \mathbb{N}$. Letting $f$ be the recursive function such that

$$f(x) = s(j, 1, x),$$

by definition of $\theta$, we get

$$\varphi_{f(x)}(y) = \theta(x, y) = \begin{cases} 
\psi(y) & \text{if } x \in K, \\
\text{undefined} & \text{if } x \notin K.
\end{cases}$$

Thus, $f$ is the desired reduction function. Now, we have

$$x \in K \iff f(x) \in P_C,$$

and thus, the characteristic function $C_K$ of $K$ is equal to $C_P \circ f$, where $C_P$ is the characteristic function of $P_C$. Therefore, $P_C$ is not recursive, since otherwise, $K$ would be recursive, a contradiction.

Rice’s Theorem shows that all nontrivial properties of the input/output behavior of programs are undecidable!

The scenario to apply Rice’s Theorem to a class $C$ of partial functions is to show that some partial recursive function belongs to $C$ ($C$ is not empty), and that some partial recursive function does not belong to $C$ ($C$ is not all the partial recursive functions). This demonstrates that $C$ is nontrivial.

In particular, the following properties are undecidable.

**Lemma 7.7.** The following properties of partial recursive functions are undecidable.

(a) A partial recursive function is a constant function.

(b) Given any integer $y \in \mathbb{N}$, is $y$ in the range of some partial recursive function.

(c) Two partial recursive functions $\varphi_x$ and $\varphi_y$ are identical.

(d) A partial recursive function $\varphi_x$ is equal to a given partial recursive function $\varphi_a$.

(e) A partial recursive function yields output $z$ on input $y$, for any given $y, z \in \mathbb{N}$.

(f) A partial recursive function diverges for some input.

(g) A partial recursive function diverges for all input.

The above Lemma is left as an easy exercise. For example, in (a), we need to exhibit a constant (partial) recursive function, such as $\text{zero}(n) = 0$, and a nonconstant (partial) recursive function, such as the identity function (or $\text{succ}(n) = n + 1$).

A property may be undecidable although it is partially decidable. By partially decidable, we mean that there exists a recursive function $g$ that enumerates the set $P_C = \{x \mid \varphi_x \in C\}$. This means that there is a recursive function $g$ whose range is $P_C$. We say that $P_C$ is *recursively enumerable*. Indeed, $g$ provides a recursive enumeration of $P_C$, with possible repetitions. Recursively enumerable sets are the object of the next Section.
7.3 Recursively Enumerable Sets

Consider the set

\[ A = \{ x \in \mathbb{N} \mid \varphi_x(a) \text{ is defined} \}, \]

where \( a \in \mathbb{N} \) is any fixed natural number. By Rice’s Theorem, \( A \) is not recursive (check this).

We claim that \( A \) is the range of a recursive function \( g \). For this, we use the \( T \)-predicate.

We produce a function which is actually primitive recursive. First, note that \( A \) is nonempty (why?), and let \( x_0 \) be any index in \( A \). We define \( g \) by primitive recursion as follows:

\[
\begin{align*}
g(0) &= x_0, \\
g(x + 1) &= \begin{cases} 
\Pi_1(x) & \text{if } T(\Pi_1(x), a, \Pi_2(x)), \\
x_0 & \text{otherwise.}
\end{cases}
\end{align*}
\]

Since this type of argument is new, it is helpful to explain informally what \( g \) does. For every input \( x \), the function \( g \) tries finitely many steps of a computation on input \( a \) of some partial recursive function. The computation is given by \( \Pi_2(x) \), and the partial function is given by \( \Pi_1(x) \). Since \( \Pi_1 \) and \( \Pi_2 \) are projection functions, when \( x \) ranges over \( \mathbb{N} \), both \( \Pi_1(x) \) and \( \Pi_2(x) \) also range over \( \mathbb{N} \).

Such a process is called a dovetailing computation. Therefore all computations on input \( a \) for all partial recursive functions will be tried, and the indices of the partial recursive functions converging on input \( a \) will be selected. This type of argument will be used over and over again.

**Definition 7.4.** A subset \( X \) of \( \mathbb{N} \) is recursively enumerable iff either \( X = \emptyset \), or \( X \) is the range of some total recursive function. Similarly, a subset \( X \) of \( \Sigma^* \) is recursively enumerable iff either \( X = \emptyset \), or \( X \) is the range of some total recursive function.

We will often abbreviate recursively enumerable as \( r.e. \) A recursively enumerable set is sometimes called a partially decidable set.

**Remark:** It should be noted that the definition of an \( r.e. \) set given in Definition 7.4 is different from an earlier definition given in terms of acceptance by a Turing machine and it is by no means obvious that these two definitions are equivalent. This equivalence will be proved in Lemma 7.9 ((1) \( \iff \) (4)).

The following Lemma relates recursive sets and recursively enumerable sets.

**Lemma 7.8.** A set \( A \) is recursive iff both \( A \) and its complement \( \overline{A} \) are recursively enumerable.

**Proof.** Assume that \( A \) is recursive. Then, it is trivial that its complement is also recursive. Hence, we only have to show that a recursive set is recursively enumerable. The empty set
is recursively enumerable by definition. Otherwise, let \( y \in A \) be any element. Then, the function \( f \) defined such that

\[
f(x) = \begin{cases} 
x & \text{iff } C_A(x) = 1, \\
y & \text{iff } C_A(x) = 0,
\end{cases}
\]

for all \( x \in \mathbb{N} \) is recursive and has range \( A \).

Conversely, assume that both \( A \) and \( \overline{A} \) are recursively enumerable. If either \( A \) or \( \overline{A} \) is empty, then \( A \) is recursive. Otherwise, let \( A = f(\mathbb{N}) \) and \( \overline{A} = g(\mathbb{N}) \), for some recursive functions \( f \) and \( g \). We define the function \( C_A \) as follows:

\[
C_A(x) = \begin{cases} 
1 & \text{if } f(\min y \{ f(y) = x \vee g(y) = x \}) = x, \\
0 & \text{otherwise}. 
\end{cases}
\]

The function \( C_A \) lists \( A \) and \( \overline{A} \) in parallel, waiting to see whether \( x \) turns up in \( A \) or in \( \overline{A} \). Note that \( x \) must eventually turn up either in \( A \) or in \( \overline{A} \), so that \( C_A \) is a total recursive function. \( \square \)

Our next goal is to show that the recursively enumerable sets can be given several equivalent definitions.

**Lemma 7.9.** For any subset \( A \) of \( \mathbb{N} \), the following properties are equivalent:

1. \( A \) is empty or \( A \) is the range of a primitive recursive function (Rosser, 1936).
2. \( A \) is recursively enumerable.
3. \( A \) is the range of a partial recursive function.
4. \( A \) is the domain of a partial recursive function.

**Proof.** The implication (1) \( \Rightarrow \) (2) is trivial, since \( A \) is r.e. iff either it is empty or it is the range of a (total) recursive function.

To prove the implication (2) \( \Rightarrow \) (3), it suffices to observe that the empty set is the range of the totally undefined function (computed by an infinite loop program), and that a recursive function is a partial recursive function.

The implication (3) \( \Rightarrow \) (4) is shown as follows. Assume that \( A \) is the range of \( \varphi_i \). Define the function \( f \) such that

\[
f(x) = \min y \{ T(i, \Pi_1(y), \Pi_2(y)) \wedge \text{Res}(\Pi_2(y)) = x \}
\]

for all \( x \in \mathbb{N} \). Clearly, \( f \) is partial recursive and has domain \( A \).

The implication (4) \( \Rightarrow \) (1) is shown as follows. The only nontrivial case is when \( A \) is nonempty. Assume that \( A \) is the domain of \( \varphi_i \). Since \( A \neq \emptyset \), there is some \( a \in \mathbb{N} \) such that \( a \in A \), so the quantity

\[
\min y \{ T(i, \Pi_1(y), \Pi_2(y)) \}
\]
is defined and we can pick \(a\) to be
\[
a = \Pi_1(\min y[T(i, \Pi_1(y), \Pi_2(y))]).
\]
We define the primitive recursive function \(f\) as follows:
\[
f(0) = a,
\]
\[
f(x + 1) = \begin{cases} 
\Pi_1(x) & \text{if } T(i, \Pi_1(x), \Pi_2(x)), \\
 a & \text{if } \neg T(i, \Pi_1(x), \Pi_2(x)).
\end{cases}
\]
Clearly, \(A\) is the range of \(f\) and \(f\) is primitive recursive.

More intuitive proofs of the implications (3) \(\Rightarrow\) (4) and (4) \(\Rightarrow\) (1) can be given. Assume that \(A \neq \emptyset\) and that \(A = \text{range}(g)\), where \(g\) is a partial recursive function. Assume that \(g\) is computed by a RAM program \(P\). To compute \(f(x)\), we start computing the sequence
\[
g(0), g(1), \ldots
\]
looking for \(x\). If \(x\) turns up as say \(g(n)\), then we output \(n\). Otherwise the computation diverges. Hence, the domain of \(f\) is the range of \(g\).

Assume now that \(A\) is the domain of some partial recursive function \(g\), and that \(g\) is computed by some Turing machine \(M\). Since the case where \(A = \emptyset\) is trivial, we may assume that \(A \neq \emptyset\), and let \(n_0 \in A\) be some chosen element in \(A\). We construct another Turing machine performing the following steps: On input \(n\),

(0) Do one step of the computation of \(g(0)\)

\[
\ldots
\]

\(n)\) Do \(n + 1\) steps of the computation of \(g(0)\)

Do \(n\) steps of the computation of \(g(1)\)

\[
\ldots
\]

Do 2 steps of the computation of \(g(n - 1)\)

Do 1 step of the computation of \(g(n)\)

During this process, whenever the computation of \(g(m)\) halts for some \(m \leq n\), we output \(m\). Otherwise, we output \(n_0\).

In this fashion, we will enumerate the domain of \(g\), and since we have constructed a Turing machine that halts for every input, we have a total recursive function.

The following Lemma can easily be shown using the proof technique of Lemma 7.9.

**Lemma 7.10.** (1) There is a recursive function \(h\) such that
\[
\text{range}(\varphi_x) = \text{dom}(\varphi_{h(x)})
\]
for all \( x \in \mathbb{N} \).

(2) There is a recursive function \( k \) such that

\[
\text{dom}(\varphi_x) = \text{range}(\varphi_{k(x)})
\]

and \( \varphi_{k(x)} \) is total recursive, for all \( x \in \mathbb{N} \) such that \( \text{dom}(\varphi_x) \neq \emptyset \).

The proof of Lemma 7.10 is left as an exercise. Using Lemma 7.9, we can prove that \( K \) is an r.e. set. Indeed, we have \( K = \text{dom}(f) \), where

\[
f(x) = \varphi_{\text{univ}}(x, x)
\]

for all \( x \in \mathbb{N} \). The set

\[
K_0 = \{ (x, y) \mid \varphi_x(y) \text{ converges} \}
\]

is also an r.e. set, since \( K_0 = \text{dom}(g) \), where

\[
g(z) = \varphi_{\text{univ}}(\Pi_1(z), \Pi_2(z)),
\]

which is partial recursive. The sets \( K \) and \( K_0 \) are examples of r.e. sets that are not recursive.

We can now prove that there are sets that are not r.e.

**Lemma 7.11.** For any indexing of the partial recursive functions, the complement \( \overline{K} \) of the set

\[
K = \{ x \in \mathbb{N} \mid \varphi_x(x) \text{ converges} \}
\]

is not recursively enumerable.

**Proof.** If \( \overline{K} \) was recursively enumerable, since \( K \) is also recursively enumerable, by Lemma 7.8, the set \( K \) would be recursive, a contradiction. \( \blacksquare \)

The sets \( \overline{K} \) and \( \overline{K_0} \) are examples of sets that are not r.e. This shows that the r.e. sets are not closed under complementation. However, we leave it as an exercise to prove that the r.e. sets are closed under union and intersection.

We will prove later on that \( \text{TOTAL} \) is not r.e. This is rather unpleasant. Indeed, this means that there is no way of effectively listing all algorithms (all total recursive functions). Hence, in a certain sense, the concept of partial recursive function (procedure) is more natural than the concept of a (total) recursive function (algorithm).

The next two Lemmas give other characterizations of the r.e. sets and of the recursive sets. The proofs are left as an exercise.

**Lemma 7.12.** (1) A set \( A \) is r.e. iff either it is finite or it is the range of an injective recursive function.

(2) A set \( A \) is r.e. if either it is empty or it is the range of a monotonic partial recursive function.

(3) A set \( A \) is r.e. iff there is a Turing machine \( M \) such that, for all \( x \in \mathbb{N} \), \( M \) halts on \( x \) iff \( x \in A \).
Lemma 7.13. A set $A$ is recursive iff either it is finite or it is the range of a strictly increasing recursive function.

Another important result relating the concept of partial recursive function and that of an r.e. set is given below.

Theorem 7.14. For every unary partial function $f$, the following properties are equivalent:

1. $f$ is partial recursive.
2. The set $\{\langle x, f(x) \rangle \mid x \in \text{dom}(f)\}$ is r.e.

Proof. Let $g(x) = \langle x, f(x) \rangle$. Clearly, $g$ is partial recursive, and

$$\text{range}(g) = \{\langle x, f(x) \rangle \mid x \in \text{dom}(f)\}.$$ 

Conversely, assume that

$$\text{range}(g) = \{\langle x, f(x) \rangle \mid x \in \text{dom}(f)\}$$

for some recursive function $g$. Then, we have

$$f(x) = \Pi_2(g(\min y[\Pi_1(g(y)) = x]))$$

for all $x \in \mathbb{N}$, so that $f$ is partial recursive. \qed

Using our indexing of the partial recursive functions and Lemma 7.9, we obtain an indexing of the r.e. sets.

Definition 7.5. For any acceptable indexing $\varphi_0, \varphi_1, \ldots$ of the partial recursive functions, we define the enumeration $W_0, W_1, \ldots$ of the r.e. sets by setting

$$W_x = \text{dom}(\varphi_x).$$

We now describe a technique for showing that certain sets are r.e. but not recursive, or complements of r.e. sets that are not recursive, or not r.e., or neither r.e. nor the complement of an r.e. set. This technique is known as reducibility.
7.4 Reducibility and Complete Sets

We already used the notion of reducibility in the proof of Lemma 7.5 to show that TOTAL is not recursive.

**Definition 7.6.** Let $A$ and $B$ be subsets of $\mathbb{N}$ (or $\Sigma^*$). We say that the set $A$ is *many-one reducible* to the set $B$ if there is a total recursive function $f: \mathbb{N} \to \mathbb{N}$ (or $f: \Sigma^* \to \Sigma^*$) such that

$$x \in A \iff f(x) \in B \quad \text{for all } x \in \mathbb{N}.$$ 

We write $A \leq B$, and for short, we say that $A$ is *reducible* to $B$.

Intuitively, deciding membership in $B$ is as hard as deciding membership in $A$. This is because any method for deciding membership in $B$ can be converted to a method for deciding membership in $A$ by first applying $f$ to the number (or string) to be tested.

The following simple Lemma is left as an exercise to the reader.

**Lemma 7.15.** Let $A, B, C$ be subsets of $\mathbb{N}$ (or $\Sigma^*$). The following properties hold:

1. If $A \leq B$ and $B \leq C$, then $A \leq C$.
2. If $A \leq B$ then $\overline{A} \leq \overline{B}$.
3. If $A \leq B$ and $B$ is r.e., then $A$ is r.e.
4. If $A \leq B$ and $A$ is not r.e., then $B$ is not r.e.
5. If $A \leq B$ and $B$ is recursive, then $A$ is recursive.
6. If $A \leq B$ and $A$ is not recursive, then $B$ is not recursive.

Another important concept is the concept of a complete set.

**Definition 7.7.** An r.e. set $A$ is *complete w.r.t. many-one reducibility* iff every r.e. set $B$ is reducible to $A$, i.e., $B \leq A$.

For simplicity, we will often say *complete* for *complete w.r.t. many-one reducibility*. Intuitively, a complete r.e. set is a “hardest” r.e. set as far as membership is concerned.

**Theorem 7.16.** The following properties hold:

1. If $A$ is complete, $B$ is r.e., and $A \leq B$, then $B$ is complete.
2. $K_0$ is complete.
3. $K_0$ is reducible to $K$. 
Proof. (1) This is left as a simple exercise.

(2) Let \( W_x \) be any r.e. set. Then
\[
y \in W_x \iff \langle x, y \rangle \in K_0,
\]
and the reduction function is the recursive function \( f \) such that
\[
f(y) = \langle x, y \rangle
\]
for all \( y \in \mathbb{N} \).

(3) We use the s-m-n Theorem. First, we leave it as an exercise to prove that there is a recursive function \( f \) such that
\[
\varphi_{f(x)}(y) = \begin{cases} 1 & \text{if } \varphi_{\Pi_1(x)}(\Pi_2(x)) \text{ converges}, \\ \text{undefined} & \text{otherwise}, \end{cases}
\]
for all \( x, y \in \mathbb{N} \). Then, for every \( z \in \mathbb{N} \),
\[
z \in K_0 \iff \varphi_{\Pi_1(z)}(\Pi_2(z)) \text{ converges},
\]
iff \( \varphi_{f(z)}(y) = 1 \) for all \( y \in \mathbb{N} \). However,
\[
\varphi_{f(z)}(y) = 1 \iff \varphi_{f(z)}(f(z)) = 1,
\]
since \( \varphi_{f(z)} \) is a constant function. This means that
\[
z \in K_0 \iff f(z) \in K,
\]
and \( f \) is the desired function.

As a corollary of Theorem 7.16, the set \( K \) is also complete.

Definition 7.8. Two sets \( A \) and \( B \) have the same degree of unsolvability or are equivalent iff \( A \leq B \) and \( B \leq A \).

Since \( K \) and \( K_0 \) are both complete, they have the same degree of unsolvability. We will now investigate the reducibility and equivalence of various sets. Recall that
\[
\text{TOTAL} = \{ x \in \mathbb{N} \mid \varphi_x \text{ is total} \}.
\]
We define \( \text{EMPTY} \) and \( \text{FINITE} \), as follows:
\[
\text{EMPTY} = \{ x \in \mathbb{N} \mid \varphi_x \text{ is undefined for all input} \},
\]
\[
\text{FINITE} = \{ x \in \mathbb{N} \mid \varphi_x \text{ is defined only for finitely many input} \}.
\]

Obviously, \( \text{EMPTY} \subset \text{FINITE} \), and since
\[
\text{FINITE} = \{ x \in \mathbb{N} \mid \varphi_x \text{ has a finite domain} \},
\]
we have
\[
\text{FINITE} = \{ x \in \mathbb{N} \mid \varphi_x \text{ has an infinite domain} \},
\]
and thus, \( \text{TOTAL} \subset \text{FINITE} \).
Lemma 7.17. We have $K_0 \leq \text{EMPTY}$.

The proof of Lemma 7.17 follows from the proof of Theorem 7.16. We also have the following Lemma.

Lemma 7.18. The following properties hold:

1. EMPTY is not r.e.
2. $\overline{\text{EMPTY}}$ is r.e.
3. $K$ and EMPTY are equivalent.
4. EMPTY is complete.

Proof. We prove (1) and (3), leaving (2) and (4) as an exercise (Actually, (2) and (4) follow easily from (3)). First, we show that $K \leq \text{EMPTY}$. By the s-m-n Theorem, there exists a recursive function $f$ such that

$$\varphi_{f(x)}(y) = \begin{cases} 
\varphi_x(x) & \text{if } \varphi_x(x) \text{ converges,} \\
\text{undefined} & \text{if } \varphi_x(x) \text{ diverges,}
\end{cases}$$

for all $x, y \in \mathbb{N}$. Note that for all $x \in \mathbb{N}$,

$$x \in K \text{ iff } f(x) \in \text{EMPTY},$$

and thus, $K \leq \text{EMPTY}$. Since $K$ is not r.e., EMPTY is not r.e.

By the s-m-n Theorem, there is a recursive function $g$ such that

$$\varphi_{g(x)}(y) = \min z[T(x, \Pi_1(z), \Pi_2(z))],$$

for all $x, y \in \mathbb{N}$. Note that

$$x \in \text{EMPTY} \text{ iff } g(x) \in K$$

for all $x \in \mathbb{N}$. Therefore, EMPTY $\leq K$, and since we just showed that $K \leq \text{EMPTY}$, the sets $K$ and EMPTY are equivalent.

Lemma 7.19. The following properties hold:

1. TOTAL and $\overline{\text{TOTAL}}$ are not r.e.
2. $\text{FINITE}$ and $\overline{\text{FINITE}}$ are not r.e.

Proof. Checking the proof of Theorem 7.16, we note that $K_0 \leq \text{TOTAL}$ and $K_0 \leq \text{FINITE}$. Hence, we get $\overline{K_0} \leq \overline{\text{TOTAL}}$ and $\overline{K_0} \leq \overline{\text{FINITE}}$, and neither $\overline{\text{TOTAL}}$ nor $\overline{\text{FINITE}}$ is r.e. If TOTAL was r.e., then there would be a recursive function $f$ such that TOTAL $= \text{range}(f)$. Define $g$ as follows:

$$g(x) = \varphi_{f(x)}(x) + 1 = \varphi_{\text{univ}}(f(x), x) + 1$$
7.5. THE RECURSION THEOREM

for all \( x \in \mathbb{N} \). Since \( f \) is total and \( \varphi_{f(x)} \) is total for all \( x \in \mathbb{N} \), the function \( g \) is total recursive. Let \( e \) be an index such that

\[
g = \varphi_{f(e)}.\]

Since \( g \) is total, \( g(e) \) is defined. Then, we have

\[
g(e) = \varphi_{f(e)}(e) + 1 = g(e) + 1,
\]
a contradiction. Hence, TOTAL is not r.e. Finally, we show that TOTAL \( \leq \) \( \text{FINITE} \). This also shows that \( \text{FINITE} \) is not r.e. By the s-m-n Theorem, there is a recursive function \( f \) such that

\[
\varphi_{f(x)}(y) = \begin{cases} 
1 & \text{if } \forall z \leq y(\varphi_z(z) \downarrow), \\
\text{undefined} & \text{otherwise},
\end{cases}
\]
for all \( x, y \in \mathbb{N} \). It is easily seen that

\[
x \in \text{TOTAL} \iff f(x) \in \text{FINITE}
\]
for all \( x \in \mathbb{N} \).

From Lemma 7.19, we have TOTAL \( \leq \) \( \text{FINITE} \). It turns out that \( \text{FINITE} \leq \text{TOTAL} \), and TOTAL and \( \text{FINITE} \) are equivalent.

**Lemma 7.20.** The sets TOTAL and \( \text{FINITE} \) are equivalent.

**Proof.** We show that \( \text{FINITE} \leq \text{TOTAL} \). By the s-m-n Theorem, there is a recursive function \( f \) such that

\[
\varphi_{f(x)}(y) = \begin{cases} 
1 & \text{if } \exists z \geq y(\varphi_z(z) \downarrow), \\
\text{undefined} & \text{if } \forall z \geq y(\varphi_z(z) \uparrow),
\end{cases}
\]
for all \( x, y \in \mathbb{N} \). It is easily seen that

\[
x \in \text{FINITE} \iff f(x) \in \text{TOTAL}
\]
for all \( x \in \mathbb{N} \). \( \square \)

We now turn to the recursion Theorem.

### 7.5 The Recursion Theorem

The recursion Theorem, due to Kleene, is a fundamental result in recursion theory. Let \( f \) be a total recursive function. Then, it turns out that there is some \( n \) such that

\[
\varphi_n = \varphi_{f(n)}.
\]
**Theorem 7.21.** (Recursion Theorem, Version 1) Let \( \varphi_0, \varphi_1, \ldots \) be any acceptable indexing of the partial recursive functions. For every total recursive function \( f \), there is some \( n \) such that
\[
\varphi_n = \varphi_{f(n)}.
\]

**Proof.** Consider the function \( \theta \) defined such that
\[
\theta(x, y) = \varphi_{\text{univ}}(\varphi_{\text{univ}}(x, x), y)
\]
for all \( x, y \in \mathbb{N} \). The function \( \theta \) is partial recursive, and there is some index \( j \) such that \( \varphi_j = \theta \). By the s-m-n Theorem, there is a recursive function \( g \) such that
\[
\varphi_g(x)(y) = \theta(x, y).
\]
Consider the function \( f \circ g \). Since it is recursive, there is some index \( m \) such that \( \varphi_m = f \circ g \).

Let
\[
n = g(m).
\]
Since \( \varphi_m \) is total, \( \varphi_m(m) \) is defined, and we have
\[
\begin{align*}
\varphi_n(y) &= \varphi_{g(m)}(y) = \theta(m, y) = \varphi_{\text{univ}}(\varphi_{\text{univ}}(m, m), y) = \varphi_{\text{univ}}(m, m)(y) \\
&= \varphi_{\varphi_m(m)}(y) = \varphi_{f \circ g}(m)(y) = \varphi_f(g(m))(y) = \varphi_f(n)(y),
\end{align*}
\]
for all \( y \in \mathbb{N} \). Therefore, \( \varphi_n = \varphi_{f(n)} \), as desired. \( \square \)

The recursion Theorem can be strengthened as follows.

**Theorem 7.22.** (Recursion Theorem, Version 2) Let \( \varphi_0, \varphi_1, \ldots \) be any acceptable indexing of the partial recursive functions. There is a total recursive function \( h \) such that for all \( x \in \mathbb{N} \), if \( \varphi_x \) is total, then
\[
\varphi_{\varphi_x(h(x))} = \varphi_h(x).
\]

**Proof.** The recursive function \( g \) obtained in the proof of Theorem 7.21 satisfies the condition
\[
\varphi_g(x) = \varphi_{\varphi_x(x)};
\]
and it has some index \( i \) such that \( \varphi_i = g \). Recall that \( c \) is a recursive composition function such that
\[
\varphi_c(x, y) = \varphi_x \circ \varphi_y.
\]
It is easily verified that the function \( h \) defined such that
\[
h(x) = g(c(x, i))
\]
for all \( x \in \mathbb{N} \) does the job. \( \square \)

A third version of the recursion Theorem is given below.
7.5. THE RECURSION THEOREM

Theorem 7.23. (Recursion Theorem, Version 3) For all \( n \geq 1 \), there is a total recursive function \( h \) of \( n + 1 \) arguments, such that for all \( x \in \mathbb{N} \), if \( \varphi_x \) is a total recursive function of \( n + 1 \) arguments, then

\[
\varphi_{\varphi_x}(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n) = \varphi_h(x,x_1,\ldots,x_n),
\]

for all \( x_1,\ldots,x_n \in \mathbb{N} \).

Proof. Let \( \theta \) be the function defined such that

\[
\theta(x,x_1,\ldots,x_n,y) = \varphi_{\varphi_x(x,x_1,\ldots,x_n)}(y) = \varphi_{\text{univ}}(\varphi_{\text{univ}}(x,x_1,\ldots,x_n),y)
\]

for all \( x,x_1,\ldots,x_n,y \in \mathbb{N} \). By the s-m-n Theorem, there is a recursive function \( g \) such that

\[
\varphi_g(x_1,\ldots,x_n) = \varphi_{\varphi_x(x_1,\ldots,x_n)}.
\]

It is easily shown that there is a recursive function \( c \) such that

\[
\varphi_{c(i,j)}(x,x_1,\ldots,x_n) = \varphi_i(\varphi_j(x,x_1,\ldots,x_n),x_1,\ldots,x_n)
\]

for any two partial recursive functions \( \varphi_i \) and \( \varphi_j \) (viewed as functions of \( n + 1 \) arguments) and all \( x,x_1,\ldots,x_n \in \mathbb{N} \). Let \( \varphi_i = g \), and define \( h \) such that

\[
h(x,x_1,\ldots,x_n) = g(c(x,i),x_1,\ldots,x_n),
\]

for all \( x,x_1,\ldots,x_n \in \mathbb{N} \). We have

\[
\varphi_h(x_1,\ldots,x_n) = \varphi_g(c(x,i),x_1,\ldots,x_n) = \varphi_{\varphi_{c(i)}(c(x,i),x_1,\ldots,x_n)},
\]

and

\[
\varphi_{\varphi_{c(i)}(c(x,i),x_1,\ldots,x_n)} = \varphi_{\varphi_x(\varphi_i(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)};
\]

\[
= \varphi_{\varphi_x(g(c(x,i),x_1,\ldots,x_n),x_1,\ldots,x_n)};
\]

\[
= \varphi_{\varphi_x(h(x,x_1,\ldots,x_n),x_1,\ldots,x_n)}.
\]

As a first application of the recursion theorem, we can show that there is an index \( n \) such that \( \varphi_n \) is the constant function with output \( n \). Loosely speaking, \( \varphi_n \) prints its own name. Let \( f \) be the recursive function such that

\[
f(x,y) = x
\]

for all \( x,y \in \mathbb{N} \). By the s-m-n Theorem, there is a recursive function \( g \) such that

\[
\varphi_g(x)(y) = f(x,y) = x
\]
for all $x, y \in \mathbb{N}$. By the recursion Theorem 7.21, there is some $n$ such that

$$\varphi_{g(n)} = \varphi_n,$$

the constant function with value $n$.

As a second application, we get a very short proof of Rice’s Theorem. Let $C$ be such that $P_C \neq \emptyset$ and $P_C \neq \mathbb{N}$, and let $j \in P_C$ and $k \in \mathbb{N} - P_C$. Define the function $f$ as follows:

$$f(x) = \begin{cases} j & \text{if } x \notin P_C, \\ k & \text{if } x \in P_C, \end{cases}$$

If $P_C$ is recursive, then $f$ is recursive. By the recursion Theorem 7.21, there is some $n$ such that

$$\varphi_{f(n)} = \varphi_n.$$  

But then, we have

$$n \in P_C \iff f(n) \notin P_C$$

by definition of $f$, and thus,

$$\varphi_{f(n)} \neq \varphi_n,$$

a contradiction. Hence, $P_C$ is not recursive.

As a third application, we prove the following Lemma.

**Lemma 7.24.** Let $C$ be a set of partial recursive functions and let

$$A = \{ x \in \mathbb{N} \mid \varphi_x \in C \}.$$  

The set $A$ is not reducible to its complement $\overline{A}$.

**Proof.** Assume that $A \leq \overline{A}$. Then, there is a recursive function $f$ such that

$$x \in A \iff f(x) \in \overline{A}$$

for all $x \in \mathbb{N}$. By the recursion Theorem, there is some $n$ such that

$$\varphi_{f(n)} = \varphi_n.$$  

But then,

$$\varphi_n \in C \iff n \in A \iff f(n) \in \overline{A} \iff \varphi_{f(n)} \in \overline{C},$$

contradicting the fact that

$$\varphi_{f(n)} = \varphi_n.$$
7.6. EXTENDED RICE THEOREM

The recursion Theorem can also be used to show that functions defined by recursive definitions other than primitive recursion are partial recursive. This is the case for the function known as Ackermann’s function, defined recursively as follows:

\[ f(0, y) = y + 1, \]
\[ f(x + 1, 0) = f(x, 1), \]
\[ f(x + 1, y + 1) = f(x, f(x + 1, y)). \]

It can be shown that this function is not primitive recursive. Intuitively, it outgrows all primitive recursive functions. However, \( f \) is recursive, but this is not so obvious. We can use the recursion Theorem to prove that \( f \) is recursive. Consider the following definition by cases:

\[ g(n, 0, y) = y + 1, \]
\[ g(n, x + 1, 0) = \varphi_{\text{univ}}(n, x, 1), \]
\[ g(n, x + 1, y + 1) = \varphi_{\text{univ}}(n, x, \varphi_{\text{univ}}(n, x + 1, y)). \]

Clearly, \( g \) is partial recursive. By the s-m-n Theorem, there is a recursive function \( h \) such that

\[ \varphi_{h(n)}(x, y) = g(n, x, y). \]

By the recursion Theorem, there is an \( m \) such that

\[ \varphi_{h(m)} = \varphi_m. \]

Therefore, the partial recursive function \( \varphi_m(x, y) \) satisfies the definition of Ackermann’s function. We showed in a previous Section that \( \varphi_m(x, y) \) is a total function, and thus, Ackermann’s function is a total recursive function.

Hence, the recursion Theorem justifies the use of certain recursive definitions. However, note that there are some recursive definitions that are only satisfied by the completely undefined function.

In the next Section, we prove the extended Rice Theorem.

7.6 Extended Rice Theorem

The extended Rice Theorem characterizes the sets of partial recursive functions \( C \) such that \( P_C \) is r.e. First, we need to discuss a way of indexing the partial recursive functions that have a finite domain. Using the uniform projection function \( \Pi \), we define the primitive recursive function \( F \) such that

\[ F(x, y) = \Pi(y + 1, \Pi_1(x) + 1, \Pi_2(x)). \]

We also define the sequence of partial functions \( P_0, P_1, \ldots \) as follows:

\[ P_x(y) = \begin{cases} 
F(x, y) - 1 & \text{if } 0 < F(x, y) \text{ and } y < \Pi_1(x) + 1, \\
\text{undefined} & \text{otherwise}.
\end{cases} \]
Lemma 7.25. Every $P_x$ is a partial recursive function with finite domain, and every partial recursive function with finite domain is equal to some $P_x$.

The proof is left as an exercise. The easy part of the extended Rice Theorem is the following lemma. Recall that given any two partial functions $f: A \to B$ and $g: A \to B$, we say that $g$ extends $f$ iff $f \subseteq g$, which means that $g(x)$ is defined whenever $f(x)$ is defined, and if so, $g(x) = f(x)$.

Lemma 7.26. Let $C$ be a set of partial recursive functions. If there is an r.e. set $A$ such that, $\varphi_x \in C$ iff there is some $y \in A$ such that $\varphi_x$ extends $P_y$, then $P_C = \{x \mid \varphi_x \in C\}$ is r.e.

Proof. Lemma 7.26 can be restated as

$$P_C = \{x \mid \exists y \in A, P_y \subseteq \varphi_x\}$$

is r.e. If $A$ is empty, so is $P_C$, and $P_C$ is r.e. Otherwise, let $f$ be a recursive function such that

$$A = \text{range}(f).$$

Let $\psi$ be the following partial recursive function:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } P_f(\Pi_2(z)) \subseteq \varphi_{\Pi_1(z)}, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

It is clear that

$$P_C = \text{range}(\psi).$$

To see that $\psi$ is partial recursive, write $\psi(z)$ as follows:

$$\psi(z) = \begin{cases} \Pi_1(z) & \text{if } \forall w \leq \Pi_1(f(\Pi_2(z))) \left[ F(f(\Pi_2(z)), w) > 0 \right. \\ \supset \varphi_{\Pi_1(z)}(w) = F(f(\Pi_2(z)), w) - 1], \\ \text{undefined} & \text{otherwise.} \end{cases}$$

To establish the converse of Lemma 7.26, we need two Lemmas.

Lemma 7.27. If $P_C$ is r.e. and $\varphi \in C$, then there is some $P_y \subseteq \varphi$ such that $P_y \in C$.

Proof. Assume that $P_C$ is r.e. and that $\varphi \in C$. By an s-m-n construction, there is a recursive function $g$ such that

$$\varphi_{g(x)}(y) = \begin{cases} \varphi(y) & \text{if } \forall z \leq y \left[ T(x, x, z) \right], \\ \text{undefined} & \text{if } \exists z \leq y \left[ T(x, x, z) \right], \end{cases}$$

for all $x, y \in \mathbb{N}$. Observe that if $x \in K$, then $\varphi_{g(x)}$ is a finite subfunction of $\varphi$, and if $x \in \overline{K}$, then $\varphi_{g(x)} = \varphi$. Assume that no finite subfunction of $\varphi$ is in $C$. Then,

$$x \in \overline{K} \text{ iff } g(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$. Since $P_C$ is r.e., $\overline{K}$ would also be r.e., a contradiction. \qed
As a corollary of Lemma 7.27, we note that TOTAL is not r.e.

**Lemma 7.28.** If $P_C$ is r.e., $\varphi \in C$, and $\varphi \subseteq \psi$, where $\psi$ is a partial recursive function, then $\psi \in C$.

**Proof.** Assume that $P_C$ is r.e. We claim that there is a recursive function $h$ such that

$$\varphi_{h(x)}(y) = \begin{cases} 
\psi(y) & \text{if } x \in K, \\
\varphi(y) & \text{if } x \in \overline{K},
\end{cases}$$

for all $x, y \in \mathbb{N}$. Assume that $\psi \notin C$. Then

$$x \in \overline{K} \iff h(x) \in P_C$$

for all $x \in \mathbb{N}$, that is, $\overline{K} \leq P_C$, a contradiction, since $P_C$ is r.e. Therefore, $\psi \in C$. To find the function $h$ we proceed as follows: Let $\varphi = \varphi_j$ and define $\Theta$ such that

$$\Theta(x, y, z) = \begin{cases} 
\varphi(y) & \text{if } T(j, y, z) \land \neg T(x, y, w), \text{ for } 0 \leq w < z \\
\psi(y) & \text{if } T(x, z) \land \neg T(j, y, w), \text{ for } 0 \leq w < z \\
\text{undefined} & \text{otherwise.}
\end{cases}$$

Observe that if $x = y = j$, then $\Theta(j, j, z)$ is multiply defined, but since $\psi$ extends $\varphi$, we get the same value $\psi(y) = \varphi(y)$, so $\Theta$ is a well defined partial function. Clearly, for all $(m, n) \in \mathbb{N}^2$, there is at most one $z \in \mathbb{N}$ so that $\Theta(x, y, z)$ is defined, so the function $\sigma$ defined by

$$\sigma(x, y) = \begin{cases} 
z & \text{if } (x, y, z) \in \text{dom}(\Theta) \\
\text{undefined} & \text{otherwise}
\end{cases}$$

is a partial recursive function. Finally, let

$$\theta(x, y) = \Theta(x, y, \sigma(x, y)),$$

a partial recursive function. It is easy to check that

$$\theta(x, y) = \begin{cases} 
\psi(y) & \text{if } x \in K, \\
\varphi(y) & \text{if } x \in \overline{K},
\end{cases}$$

for all $x, y \in \mathbb{N}$. By the s-m-n Theorem, there is a recursive function $h$ such that

$$\varphi_{h(x)}(y) = \theta(x, y)$$

for all $x, y \in \mathbb{N}$.\hfill \qed

Observe that Lemma 7.28 yields a new proof that $\overline{\text{TOTAL}}$ is not r.e. Finally, we can prove the extended Rice Theorem.

**Theorem 7.29.** (Extended Rice Theorem) The set $P_C$ is r.e. iff there is an r.e. set $A$ such that

$$\varphi_x \in C \iff \exists y \in A (P_y \subseteq \varphi_x).$$
Proof. Let $P_C = \text{dom}(\varphi_i)$. Using the s-m-n Theorem, there is a recursive function $k$ such that

$$\varphi_{k(y)} = P_y$$

for all $y \in \mathbb{N}$. Define the r.e. set $A$ such that

$$A = \text{dom}(\varphi_i \circ k).$$

Then,

$$y \in A \iff \varphi_i(k(y)) \downarrow \iff P_y \in C.$$ 

Next, using Lemma 7.27 and Lemma 7.28, it is easy to see that

$$\varphi_x \in C \iff \exists y \in A (P_y \subseteq \varphi_x).$$

Indeed, if $\varphi_x \in C$, by Lemma 7.27, there is a finite subfunction $P_y \subseteq \varphi_x$ such that $P_y \in C$, but

$$P_y \in C \iff y \in A,$$

as desired. On the other hand, if

$$P_y \subseteq \varphi_x$$

for some $y \in A$, then

$$P_y \in C,$$

and by Lemma 7.28, since $\varphi_x$ extends $P_y$, we get

$$\varphi_x \in C.$$ 

\[\square\]

### 7.7 Creative and Productive Sets

In this section, we discuss some special sets that have important applications in logic: creative and productive sets. The concepts to be described are illustrated by the following situation. Assume that

$$W_x \subseteq \overline{K}$$

for some $x \in \mathbb{N}$. We claim that

$$x \in \overline{K} - W_x.$$ 

Indeed, if $x \in W_x$, then $\varphi_x(x)$ is defined, and by definition of $K$, we get $x \notin \overline{K}$, a contradiction. Therefore, $\varphi_x(x)$ must be undefined, that is,

$$x \in \overline{K} - W_x.$$ 

The above situation can be generalized as follows.
Definition 7.9. A set $A$ is \textit{productive} iff there is a total recursive function $f$ such that

$$
\text{if } W_x \subseteq A \text{ then } f(x) \in A - W_x
$$

for all $x \in \mathbb{N}$. The function $f$ is called the \textit{productive function of $A$}. A set $A$ is \textit{creative} if it is r.e. and if its complement $\overline{A}$ is productive.

As we just showed, $K$ is creative and $\overline{K}$ is productive. The following facts are immediate consequences of the definition.

1. A productive set is not r.e.
2. A creative set is not recursive.

Creative and productive sets arise in logic. The set of theorems of a logical theory is often creative. For example, the set of theorems in Peano’s arithmetic is creative. This yields incompleteness results.

Lemma 7.30. If a set $A$ is productive, then it has an infinite r.e. subset.

Proof. We first give an informal proof. Let $f$ be the recursive productive function of $A$. We define a recursive function $g$ as follows: Let $x_0$ be an index for the empty set, and let

$$g(0) = f(x_0).$$

Assuming that

$$\{g(0), g(1), \ldots, g(y)\}$$

is known, let $x_{y+1}$ be an index for this finite set, and let

$$g(y+1) = f(x_{y+1}).$$

Since $W_{x_{y+1}} \subseteq A$, we have $f(x_{y+1}) \in A$.

For the formal proof, we use the following facts whose proof is left as an exercise:

1. There is a recursive function $u$ such that

$$W_{u(x,y)} = W_x \cup W_y.$$

2. There is a recursive function $t$ such that

$$W_{t(x)} = \{x\}.$$

Letting $x_0$ be an index for the empty set, we define the function $h$ as follows:

$$h(0) = x_0,$$

$$h(y + 1) = u(t(f(y)), h(y)).$$

We define $g$ such that

$$g = f \circ h.$$

It is easily seen that $g$ does the job.
Another important property of productive sets is the following.

**Lemma 7.31.** If a set \( A \) is productive, then \( \overline{K} \leq A \).

**Proof.** Let \( f \) be a productive function for \( A \). Using the s-m-n Theorem, we can find a recursive function \( h \) such that

\[
W_{h(y,x)} = \begin{cases} 
\{ f(y) \} & \text{if } x \in K, \\
\emptyset & \text{if } x \in \overline{K}.
\end{cases}
\]

The above can be restated as follows:

\[
\varphi_{h(y,x)}(z) = \begin{cases} 
1 & \text{if } x \in K \text{ and } z = f(y), \\
\text{undefined} & \text{if } x \in \overline{K},
\end{cases}
\]

for all \( x, y, z \in \mathbb{N} \). By the third version of the recursion Theorem (Theorem 7.23), there is a recursive function \( g \) such that

\[
W_g(x) = W_{h(g(x),x)}
\]

for all \( x \in \mathbb{N} \). Let

\[
k = f \circ g.
\]

We claim that

\[
x \in \overline{K} \text{ iff } k(x) \in A
\]

for all \( x \in \mathbb{N} \). The verification of this fact is left as an exercise. Thus, \( \overline{K} \leq A \). \( \square \)

Using Lemma 7.31, the following fact can be shown.

**Lemma 7.32.** The following facts hold.

1. If \( A \) is productive and \( A \leq B \), then \( B \) is productive.
2. \( A \) is creative iff \( A \) is equivalent to \( K \).
3. \( A \) is creative iff \( A \) is complete,
Chapter 8

The Post Correspondence Problem; Applications to Undecidability

Results

8.1 The Post Correspondence Problem

The Post correspondence problem (due to Emil Post) is another undecidable problem that turns out to be a very helpful tool for proving problems in logic or in formal language theory to be undecidable.

Let \( \Sigma \) be an alphabet with at least two letters. An instance of the Post Correspondence problem (for short, PCP) is given by two sequences \( U = (u_1, \ldots, u_m) \) and \( V = (v_1, \ldots, v_m) \), of strings \( u_i, v_i \in \Sigma^* \).

The problem is to find whether there is a (finite) sequence \( (i_1, \ldots, i_p) \), with \( i_j \in \{1, \ldots, m\} \) for \( j = 1, \ldots, p \), so that

\[
\begin{align*}
    u_{i_1} u_{i_2} \cdots u_{i_p} &= v_{i_1} v_{i_2} \cdots v_{i_p}.
\end{align*}
\]

Equivalently, an instance of the PCP is a sequence of pairs

\[
(\underline{u_1, v_1}), \ldots, (\underline{u_m, v_m}).
\]

For example, consider the following problem:

\[
(abab, ababaaa), (aaabbb, bb), (aab, baab), (ba, baa), (ab, ba), (aa, a).
\]

There is a solution for the string 1234556:

\[
abab aaabbb aab ba ab ab aa = ababaaa bb baab ba ba ab a.
\]

We are beginning to suspect that this is a hard problem. Indeed, it is undecidable!
Theorem 8.1. (Emil Post, 1946) The Post correspondence problem is undecidable, provided that the alphabet $\Sigma$ has at least two symbols.

There are several ways of proving Theorem 8.1, but the strategy is more or less the same: Reduce the halting problem to the PCP, by encoding sequences of ID’s as partial solutions of the PCP.

For instance, this can be done for RAM programs. The first step is to show that every RAM program can be simulated by a single register RAM program.

Then, the halting problem for RAM programs with one register is reduced to the PCP (using the fact that only four kinds of instructions are needed). A proof along these lines was given by Dana Scott.

8.2 Some Undecidability Results for CFG’s

Theorem 8.2. It is undecidable whether a context-free grammar is ambiguous.

Proof. We reduce the PCP to the ambiguity problem for CFG’s. Given any instance $U = (u_1, \ldots, u_m)$ and $V = (v_1, \ldots, v_m)$ of the PCP, let $c_1, \ldots, c_m$ be $m$ new symbols, and consider the following languages:

$$L_U = \{u_{i_1} \cdots u_{i_p}c_{i_p} \cdots c_{i_1} \mid 1 \leq i_j \leq m, \quad 1 \leq j \leq p, \quad p \geq 1\}$$

$$L_V = \{v_{i_1} \cdots v_{i_p}c_{i_p} \cdots c_{i_1} \mid 1 \leq i_j \leq m, \quad 1 \leq j \leq p, \quad p \geq 1\}$$

and $L_{U,V} = L_U \cup L_V$.

We can easily construct a CFG, $G_{U,V}$, generating $L_{U,V}$. The productions are:

$$S \rightarrow S_U$$

$$S \rightarrow S_V$$

$$S_U \rightarrow u_iS_Uc_i$$

$$S_U \rightarrow u_ic_i$$

$$S_V \rightarrow v_iS_Vc_i$$

$$S_V \rightarrow v_ic_i.$$  

It is easily seen that the PCP for $(U, V)$ has a solution iff $L_U \cap L_V \neq \emptyset$ iff $G$ is ambiguous. $\square$
Remark: As a corollary, we also obtain the following result: It is undecidable for arbitrary context-free grammars $G_1$ and $G_2$ whether $L(G_1) \cap L(G_2) = \emptyset$ (see also Theorem 8.4).

Recall that the computations of a Turing Machine, $M$, can be described in terms of instantaneous descriptions, $upav$.

We can encode computations

$$ID_0 \vdash ID_1 \vdash \cdots \vdash ID_n$$

halting in a proper ID, as the language, $L_M$, consisting all of strings

$$w_0 \# w_1^R \# w_2 \# w_3^R \# \cdots \# w_{2k} \# w_{2k+1}^R,$$

or

$$w_0 \# w_1^R \# w_2 \# w_3^R \# \cdots \# w_{2k-2} \# w_{2k-1}^R \# w_{2k},$$

where $k \geq 0$, $w_0$ is a starting ID, $w_i \vdash w_{i+1}$ for all $i$ with $0 \leq i < 2k + 1$ and $w_{2k+1}$ is proper halting ID in the first case, $0 \leq i < 2k$ and $w_{2k}$ is proper halting ID in the second case.

The language $L_M$ turns out to be the intersection of two context-free languages $L^0_M$ and $L^1_M$ defined as follows:

1. The strings in $L^0_M$ are of the form

$$w_0 \# w_1^R \# w_2 \# w_3^R \# \cdots \# w_{2k} \# w_{2k+1}^R,$$

or

$$w_0 \# w_1^R \# w_2 \# w_3^R \# \cdots \# w_{2k-2} \# w_{2k-1}^R \# w_{2k},$$

where $w_{2i} \vdash w_{2i+1}$ for all $i \geq 0$, and $w_{2k}$ is a proper halting ID in the second case.

2. The strings in $L^1_M$ are of the form

$$w_0 \# w_1^R \# w_2 \# w_3^R \# \cdots \# w_{2k} \# w_{2k+1}^R,$$

or

$$w_0 \# w_1^R \# w_2 \# w_3^R \# \cdots \# w_{2k-2} \# w_{2k-1}^R \# w_{2k},$$

where $w_{2i+1} \vdash w_{2i+2}$ for all $i \geq 0$, $w_0$ is a starting ID, and $w_{2k+1}$ is a proper halting ID in the first case.

**Theorem 8.3.** Given any Turing machine $M$, the languages $L^0_M$ and $L^1_M$ are context-free, and $L_M = L^0_M \cap L^1_M$.

**Proof.** We can construct PDA’s accepting $L^0_M$ and $L^1_M$. It is easily checked that $L_M = L^0_M \cap L^1_M$.

As a corollary, we obtain the following undecidability result:
Theorem 8.4. It is undecidable for arbitrary context-free grammars $G_1$ and $G_2$ whether $L(G_1) \cap L(G_2) = \emptyset$.

Proof. We can reduce the problem of deciding whether a partial recursive function is undefined everywhere to the above problem. By Rice’s theorem, the first problem is undecidable.

However, this problem is equivalent to deciding whether a Turing machine never halts in a proper ID. By Theorem 8.3, the languages $L_M^0$ and $L_M^1$ are context-free. Thus, we can construct context-free grammars $G_1$ and $G_2$ so that $L_M^0 = L(G_1)$ and $L_M^1 = L(G_2)$. Then, $M$ never halts in a proper ID iff $L_M = \emptyset$ iff (by Theorem 8.3), $L_M = L(G_1) \cap L(G_2) = \emptyset$. □

Given a Turing machine $M$, the language $L_M$ is defined over the alphabet $\Delta = \Gamma \cup Q \cup \{\#\}$. The following fact is also useful to prove undecidability:

Theorem 8.5. Given any Turing machine $M$, the language $\Delta^* - L_M$ is context-free.

Proof. One can easily check that the conditions for not belonging to $L_M$ can be checked by a PDA. □

As a corollary, we obtain:

Theorem 8.6. Given any context-free grammar, $G = (V, \Sigma, P, S)$, it is undecidable whether $L(G) = \Sigma^*$.

Proof. We can reduce the problem of deciding whether a Turing machine never halts in a proper ID to the above problem.

Indeed, given $M$, by Theorem 8.5, the language $\Delta^* - L_M$ is context-free. Thus, there is a CFG, $G$, so that $L(G) = \Delta^* - L_M$. However, $M$ never halts in a proper ID iff $L_M = \emptyset$ iff $L(G) = \Delta^*$. □

As a consequence, we also obtain the following:

Theorem 8.7. Given any two context-free grammar, $G_1$ and $G_2$, and any regular language, $R$, the following facts hold:

1. $L(G_1) = L(G_2)$ is undecidable.
2. $L(G_1) \subseteq L(G_2)$ is undecidable.
3. $L(G_1) = R$ is undecidable.
4. $R \subseteq L(G_2)$ is undecidable.

In contrast to (4), the property $L(G_1) \subseteq R$ is decidable!
8.3 More Undecidable Properties of Languages; Greibach’s Theorem

We conclude with a nice theorem of S. Greibach, which is a sort of version of Rice’s theorem for families of languages.

Let $\mathcal{L}$ be a countable family of languages. We assume that there is a coding function $c: \mathcal{L} \rightarrow \mathbb{N}$ and that this function can be extended to code the regular languages (all alphabets are subsets of some given countably infinite set).

We also assume that $\mathcal{L}$ is effectively closed under union, and concatenation with the regular languages.

This means that given any two languages $L_1$ and $L_2$ in $\mathcal{L}$, we have $L_1 \cup L_2 \in \mathcal{L}$, and $c(L_1 \cup L_2)$ is given by a recursive function of $c(L_1)$ and $c(L_2)$, and that for every regular language $R$, we have $L_1 R \in \mathcal{L}$, $RL_1 \in \mathcal{L}$, and $c(RL_1)$ and $c(L_1 R)$ are recursive functions of $c(R)$ and $c(L_1)$.

Given any language, $L \subseteq \Sigma^*$, and any string, $w \in \Sigma^*$, we define $L/w$ by

$$L/w = \{ u \in \Sigma^* \mid uw \in L \}.$$

**Theorem 8.8. (Greibach)** Let $\mathcal{L}$ be a countable family of languages that is effectively closed under union, and concatenation with the regular languages, and assume that the problem $L = \Sigma^*$ is undecidable for $L \in \mathcal{L}$ and any given sufficiently large alphabet $\Sigma$. Let $P$ be any nontrivial property of languages that is true for the regular languages, and so that if $P(L)$ holds for any $L \in \mathcal{L}$, then $P(L/a)$ also holds for any letter $a$. Then, $P$ is undecidable for $\mathcal{L}$.

**Proof.** Since $P$ is nontrivial for $\mathcal{L}$, there is some $L_0 \in \mathcal{L}$ so that $P(L_0)$ is false.

Let $\Sigma$ be large enough, so that $L_0 \subseteq \Sigma^*$, and the problem $L = \Sigma^*$ is undecidable for $L \in \mathcal{L}$.

We show that given any $L \in \mathcal{L}$, with $L \subseteq \Sigma^*$, we can construct a language $L_1 \in \mathcal{L}$, so that $L = \Sigma^*$ iff $P(L_1)$ holds. Thus, the problem $L = \Sigma^*$ for $L \in \mathcal{L}$ reduces to property $P$ for $\mathcal{L}$, and since for $\Sigma$ big enough, the first problem is undecidable, so is the second.

For any $L \in \mathcal{L}$, with $L \subseteq \Sigma^*$, let

$$L_1 = L_0 \# \Sigma^* \cup \Sigma^* \# L.$$

Since $\mathcal{L}$ is effectively closed under union and concatenation with the regular languages, we have $L_1 \in \mathcal{L}$.

If $L = \Sigma^*$, then $L_1 = \Sigma^* \# \Sigma^*$, a regular language, and thus, $P(L_1)$ holds, since $P$ holds for the regular languages.

Conversely, we would like to prove that if $L \neq \Sigma^*$, then $P(L_1)$ is false.
Since $L \neq \Sigma^*$, there is some $w \notin L$. But then,

$$L_1/\#w = L_0.$$ 

Since $P$ is preserved under quotient by a single letter, by a trivial induction, if $P(L_1)$ holds, then $P(L_0)$ also holds. However, $P(L_0)$ is false, so $P(L_1)$ must be false.

Thus, we proved that $L = \Sigma^*$ iff $P(L_1)$ holds, as claimed. 

Greibach’s theorem can be used to show that it is undecidable whether a context-free grammar generates a regular language.

It can also be used to show that it is undecidable whether a context-free language is inherently ambiguous.
Chapter 9

Computational Complexity; \( \mathcal{P} \) and \( \mathcal{NP} \)

9.1 The Class \( \mathcal{P} \)

In the previous two chapters, we clarified what it means for a problem to be decidable or undecidable. This chapter is heavily inspired by Lewis and Papadimitriou’s excellent treatment [1].

In principle, if a problem is decidable, then there is an algorithm (i.e., a procedure that halts for every input) that decides every instance of the problem.

However, from a practical point of view, knowing that a problem is decidable may be useless, if the number of steps (\textit{time complexity}) required by the algorithm is excessive, for example, exponential in the size of the input, or worse.

For instance, consider the \textit{traveling salesman problem}, which can be formulated as follows:

We have a set \( \{c_1, \ldots, c_n\} \) of cities, and an \( n \times n \) matrix \( D = (d_{ij}) \) of nonnegative integers, the \textit{distance matrix}, where \( d_{ij} \) denotes the distance between \( c_i \) and \( c_j \), which means that \( d_{ii} = 0 \) and \( d_{ij} = d_{ji} \) for all \( i \neq j \).

The problem is to find a \textit{shortest tour} of the cities, that is, a permutation \( \pi \) of \( \{1, \ldots, n\} \) so that the \textit{cost}

\[
C(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)}
\]

is as small as possible (minimal).

One way to solve the problem is to consider all possible tours, i.e., \( n! \) permutations.

Actually, since the starting point is irrelevant, we need only consider \((n - 1)!\) tours, but this still grows very fast. For example, when \( n = 40 \), it turns out that \( 39! \) exceeds \( 10^{45} \), a huge number.
Consider the $4 \times 4$ symmetric matrix given by
\[
D = \begin{pmatrix}
0 & 2 & 1 & 1 \\
2 & 0 & 1 & 1 \\
1 & 1 & 0 & 3 \\
1 & 1 & 3 & 0
\end{pmatrix},
\]
and the budget $B = 4$. The tour specified by the permutation
\[
\pi = \begin{pmatrix}
1 & 2 & 3 & 4 \\
1 & 4 & 2 & 3
\end{pmatrix}
\]
has cost 4, since
\[
c(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)}
= d_{14} + d_{42} + d_{23} + d_{31}
= 1 + 1 + 1 + 1 = 4.
\]
The cities in this tour are traversed in the order
\[
(1, 4, 2, 3, 1).
\]

Thus, to capture the essence of practically feasible algorithms, we must limit our computational devices to run only for a number of steps that is bounded by a polynomial in the length of the input.

We are led to the definition of polynomially bounded computational models.

**Definition 9.1.** A deterministic Turing machine $M$ is said to be **polynomially bounded** if there is a polynomial $p(X)$ so that the following holds: For every input $x \in \Sigma^*$, there is no ID $ID_n$ so that
\[
ID_0 \vdash ID_1 \vdash \ldots ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|),
\]
where $ID_0 = q_0x$ is the starting ID.

A language $L \subseteq \Sigma^*$ is **polynomially decidable** if there is a polynomially bounded Turing machine that accepts $L$. The family of all polynomially decidable languages is denoted by $\mathcal{P}$.

**Remark:** Even though Definition 9.1 is formulated for Turing machines, it can also be formulated for other models, such as RAM programs.

The reason is that the conversion of a Turing machine into a RAM program (and vice versa) produces a program (or a machine) whose size is polynomial in the original device.

The following lemma, although trivial, is useful:
Lemma 9.1. The class $\mathcal{P}$ is closed under complementation.

Of course, many languages do not belong to $\mathcal{P}$. One way to obtain such languages is to use a diagonal argument. But there are also many natural languages that are not in $\mathcal{P}$, although this may be very hard to prove for some of these languages.

Let us consider a few more problems in order to get a better feeling for the family $\mathcal{P}$.

9.2 Directed Graphs, Paths

Recall that a directed graph, $G$, is a pair $G = (V, E)$, where $E \subseteq V \times V$. Every $u \in V$ is called a node (or vertex) and a pair $(u, v) \in E$ is called an edge of $G$.

We will restrict ourselves to simple graphs, that is, graphs without edges of the form $(u, u)$; equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a path from $u$ to $v$ is any sequence of $n + 1$ edges ($n \geq 0$)

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from $u$ to $v$ is simply a single edge, $(u, v)$.)

A graph $G$ is strongly connected if for every pair $(u, v) \in V \times V$, there is a path from $u$ to $v$. A closed path, or cycle, is a path from some node $u$ to itself.

We will restrict our attention to finite graphs, i.e. graphs $(V, E)$ where $V$ is a finite set.

Definition 9.2. Given a graph $G$, an Eulerian cycle is a cycle in $G$ that passes through all the nodes (possibly more than once) and every edge of $G$ exactly once. A Hamiltonian cycle is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Eulerian Cycle Problem: Given a graph $G$, is there an Eulerian cycle in $G$?

Hamiltonian Cycle Problem: Given a graph $G$, is there an Hamiltonian cycle in $G$?

9.3 Eulerian Cycles

The following graph is a directed graph version of the Königsberg bridge problem, solved by Euler in 1736.

The nodes $A, B, C, D$ correspond to four areas of land in Königsberg and the edges to the seven bridges joining these areas of land.
The problem is to find a closed path that crosses every bridge exactly once and returns to the starting point.

In fact, the problem is unsolvable, as shown by Euler, because some nodes do not have the same number of incoming and outgoing edges (in the undirected version of the problem, some nodes do not have an even degree.)

It may come as a surprise that the Eulerian Cycle Problem does have a polynomial time algorithm, but so far, not such algorithm is known for the Hamiltonian Cycle Problem.

The reason why the Eulerian Cycle Problem is decidable in polynomial time is the following theorem due to Euler:

**Theorem 9.2.** A graph \( G = (V,E) \) has an Eulerian cycle iff the following properties hold:

\begin{enumerate}
  \item The graph \( G \) is strongly connected.
  \item Every node has the same number of incoming and outgoing edges.
\end{enumerate}

Proving that properties (1) and (2) hold if \( G \) has an Eulerian cycle is fairly easy. The converse is harder, but not that bad (try!).

Theorem 9.2 shows that it is necessary to check whether a graph is strongly connected. This can be done by computing the transitive closure of \( E \), which can be done in polynomial time (in fact, \( O(n^3) \)).

Checking property (2) can clearly be done in polynomial time. Thus, the Eulerian cycle problem is in \( \mathcal{P} \).

Unfortunately, no theorem analogous to Theorem 9.2 is known for Hamiltonian cycles.
9.4 Hamiltonian Cycles

A game invented by Sir William Hamilton in 1859 uses a regular solid dodecahedron whose twenty vertices are labeled with the names of famous cities.

The player is challenged to “travel around the world” by finding a closed cycle along the edges of the dodecahedron which passes through every city exactly once (this is the undirected version of the Hamiltonian cycle problem).

In graphical terms, assuming an orientation of the edges between cities, the graph $D$ shown in Figure 9.2 is a plane projection of a regular dodecahedron and we want to know if there is a Hamiltonian cycle in this directed graph.

Figure 9.2: A tour “around the world.”

Finding a Hamiltonian cycle in this graph does not appear to be so easy!

A solution is shown in Figure 9.3 below:
CHAPTER 9. COMPUTATIONAL COMPLEXITY; \( \mathcal{P} \) AND \( \mathcal{NP} \)

Figure 9.3: A Hamiltonian cycle in \( D \).

A solution!

**Remark:** We talked about problems being decidable in polynomial time. Obviously, this is equivalent to deciding some property of a certain class of objects, for example, finite graphs.

Our framework requires that we first encode these classes of objects as strings (or numbers), since \( \mathcal{P} \) consists of languages.

Thus, when we say that a property is decidable in polynomial time, we are really talking about the encoding of this property as a language. Thus, we have to be careful about these encodings, but it is rare that encodings cause problems.

### 9.5 Propositional Logic and Satisfiability

We define the syntax and the semantics of propositions in conjunctive normal form (CNF).

The syntax has to do with the legal form of propositions in CNF. Such propositions are interpreted as truth functions, by assigning truth values to their variables.

We begin by defining propositions in CNF. Such propositions are constructed from a countable set, \( \mathbf{PV} \), of *propositional (or boolean) variables*, say

\[
\mathbf{PV} = \{x_1, x_2, \ldots, \}
\]
9.5. PROPOSITIONAL LOGIC AND SATISFIABILITY

using the connectives \( \land \) (and), \( \lor \) (or) and \( \neg \) (negation).

We define a literal (or atomic proposition), \( L \), as \( L = x \) or \( L = \neg x \), also denoted by \( \overline{x} \), where \( x \in PV \).

A clause, \( C \), is a disjunction of pairwise distinct literals,

\[
C = (L_1 \lor L_2 \lor \cdots \lor L_m)
\]

Thus, a clause may also be viewed as a nonempty set

\[
C = \{L_1, L_2, \ldots, L_m\}
\]

We also have a special clause, the empty clause, denoted \( \bot \) or \( \Box \) (or \( \{} \)). It corresponds to the truth value false.

A proposition in CNF, or boolean formula, \( P \), is a conjunction of pairwise distinct clauses

\[
P = C_1 \land C_2 \land \cdots \land C_n
\]

Thus, a boolean formula may also be viewed as a nonempty set

\[
P = \{C_1, \ldots, C_n\}
\]

but this time, the comma is interpreted as conjunction. We also allow the proposition \( \bot \), and sometimes the proposition \( \top \) (corresponding to the truth value true).

A truth assignment (or valuation), \( v \), is any function \( v : PV \to BOOL \).

Given a truth assignment, \( v : PV \to BOOL \), we define the truth value, \( \hat{v}(X) \), of a literal, clause, and boolean formula, \( X \), using the following recursive definition:

1. \( \hat{v}(\bot) = F \), \( \hat{v}(\top) = T \).
2. \( \hat{v}(x) = v(x) \), if \( x \in PV \).
3. \( \hat{v}(\overline{x}) = \overline{v(x)} \), if \( x \in PV \), where \( \overline{v(x)} = F \) if \( v(x) = T \) and \( \overline{v(x)} = T \) if \( v(x) = F \).
(4) \( \hat{v}(C) = F \) if \( C \) is a clause and iff \( \hat{v}(L_i) = F \) for all literals \( L_i \) in \( C \), otherwise \( T \).

(5) \( \hat{v}(P) = T \) if \( P \) is a boolean formula and iff \( \hat{v}(C_j) = T \) for all clauses \( C_j \) in \( P \), otherwise \( F \).

**Definition 9.3.** We say that a truth assignment, \( v \), *satisfies* a boolean formula, \( P \), if \( \hat{v}(P) = T \). In this case, we also write
\[
v \models P.
\]
A boolean formula, \( P \), is *satisfiable* if \( v \models P \) for some truth assignment \( v \), otherwise, it is *unsatisfiable*. A boolean formula, \( P \), is *valid (or a tautology)* if \( v \models P \) for all truth assignments \( v \), in which case we write
\[
\models P.
\]

One should check that the boolean formula
\[
P = \{(x_1 \lor x_2 \lor x_3), (\overline{x}_1 \lor x_2), (\overline{x}_2 \lor x_3), (\overline{x}_3 \lor x_1), (\overline{x}_1 \lor \overline{x}_2 \lor \overline{x}_3)\}
\]
is unsatisfiable.

One may think that it is easy to test whether a proposition is satisfiable or not. Try it, it is not that easy!

As a matter of fact, the *satisfiability problem*, testing whether a boolean formula is satisfiable, also denoted SAT, is not known to be in \( \mathcal{P} \).

Moreover, it is an NP-complete problem. Most people believe that the satisfiability problem is not in \( \mathcal{P} \), but a proof still eludes us!

Before we explain what is the class \( \mathcal{NP} \), let us remark that the satisfiability problem for clauses containing at most two literals (2-satisfiability, or 2-SAT) is solvable in polynomial time.

The first step consists in observing that if every clause in \( P \) contains at most two literals, then we can reduce the problem to testing satisfiability when every clause has exactly two literals.

Indeed, if \( P \) contains some clause \((x)\), then any valuation satisfying \( P \) must make \( x \) true. Then, all clauses containing \( x \) will be true, and we can delete them, whereas we can delete \( \overline{x} \) from every clause containing it, since \( \overline{x} \) is false.

Similarly, if \( P \) contains some clause \((\overline{x})\), then any valuation satisfying \( P \) must make \( x \) false.

Thus, in a finite number of steps, either we get the empty clause, and \( P \) is unsatisfiable, or we get a set of clauses with exactly two literals.

The number of steps is clearly linear in the number of literals in \( P \).

For the second step, we construct a directed graph from \( P \). The nodes of this graph are the literals in \( P \), and edges are defined as follows:
9.6. THE CLASS NP, NP-COMPLETENESS

(1) For every clause \((x \lor y)\), there is an edge from \(x\) to \(y\) and an edge from \(\overline{y}\) to \(x\).

(2) For every clause \((x \lor y)\), there is an edge from \(x\) to \(y\) and an edge from \(\overline{y}\) to \(x\).

(3) For every clause \((\overline{x} \lor \overline{y})\), there is an edge from \(x\) to \(\overline{y}\) and an edge from \(y\) to \(\overline{x}\).

Then, it can be shown that \(P\) is unsatisfiable if and only if there is some \(x\) so that there is a cycle containing \(x\) and \(\overline{x}\).

As a consequence, 2-satisfiability is in \(P\).

9.6 The Class \(\text{NP}\), Polynomial Reducibility, \(\text{NP}\)-Completeness

One will observe that the hard part in trying to solve either the Hamiltonian cycle problem or the satisfiability problem, SAT, is to find a solution, but that checking that a candidate solution is indeed a solution can be done easily in polynomial time.

This is the essence of problems that can be solved nondeterministically in polynomial time: A solution can be guessed and then checked in polynomial time.

Definition 9.4. A nondeterministic Turing machine \(M\) is said to be polynomially bounded if there is a polynomial \(p(X)\) so that the following holds: For every input \(x \in \Sigma^*\), there is no ID \(ID_n\) so that

\[ ID_0 \vdash ID_1 \vdash^* ID_{n-1} \vdash ID_n, \quad \text{with} \quad n > p(|x|), \]

where \(ID_0 = q_0x\) is the starting ID.

A language \(L \subseteq \Sigma^*\) is nondeterministic polynomially decidable if there is a polynomially bounded nondeterministic Turing machine that accepts \(L\). The family of all nondeterministic polynomially decidable languages is denoted by \(\text{NP}\).

Of course, we have the inclusion \(\text{P} \subseteq \text{NP}\), but whether or not we have equality is one of the most famous open problems of theoretical computer science and mathematics.

In fact, the question \(\text{P} \neq \text{NP}\) is one of the open problems listed by the CLAY Institute, together with the Poincaré conjecture and the Riemann hypothesis, among other problems, and for which one million dollar is offered as a reward!

It is easy to check that SAT is in \(\text{NP}\), and so is the Hamiltonian cycle problem.

As we saw in recursion theory, where we introduced the notion of many-one reducibility, in order to compare the “degree of difficulty” of problems, it is useful to introduce the notion of reducibility and the notion of a complete set.
Definition 9.5. A function $f: \Sigma^* \rightarrow \Sigma^*$ is polynomial-time computable if there is a polynomial $p(X)$ so that the following holds: There is a deterministic Turing machine $M$ computing it so that for every input $x \in \Sigma^*$, there is no ID $ID_n$ so that

$$ID_0 \vdash ID_1 \vdash \cdots \vdash ID_{n-1} \vdash ID_n, \text{ with } n > p(|x|),$$

where $ID_0 = q_0 x$ is the starting ID.

Given two languages $L_1, L_2 \subseteq \Sigma^*$, a polynomial reduction from $L_1$ to $L_2$ is a polynomial-time computable function $f: \Sigma^* \rightarrow \Sigma^*$ so that for all $u \in \Sigma^*$,

$$u \in L_1 \iff f(u) \in L_2.$$

A polynomial reduction $f: \Sigma^* \rightarrow \Sigma^*$ from a language $L_1$ to a language $L_2$ is a method that converts in polynomial time every string $u \in \Sigma^*$ (viewed as an instance of a problem $A$ encoded by language $L_1$) to a string $f(u) \in \Sigma^*$ (viewed as an instance of a problem $B$ encoded by language $L_2$) in such way that membership in $L_1$, that is $u \in L_1$, is equivalent to membership in $L_2$, that is $f(u) \in L_2$.

As a consequence, if we have a procedure to decide membership in $L_2$ (to solve every instance of problem $B$), then we have a procedure for solving membership in $L_1$ (to solve every instance of problem $A$), since given any $u \in L_1$, we can first apply $f$ to $u$ to produce $f(u)$, and then apply our procedure to decide whether $f(u) \in L_2$; the defining property of $f$ says that this is equivalent to deciding whether $u \in L_1$. Furthermore, if the procedure for deciding membership in $L_2$ runs deterministically in polynomial time, since $f$ runs deterministically in polynomial time, so does the procedure for deciding membership in $L_1$, and similarly if the procedure for deciding membership in $L_2$ runs non deterministically in polynomial time.

For the above reason, we see that membership in $L_2$ can be considered at least as hard as membership in $L_1$, since any method for deciding membership in $L_2$ yields a method for deciding membership in $L_1$. Thus, if we view $L_1$ an encoding a problem $A$ and $L_2$ as encoding a problem $B$, then $B$ is at least as hard as $A$.

Intuitively, we also see that if $L_1$ is a hard problem and $L_1$ can be reduced to $L_2$, then $L_2$ is also a hard problem.

For example, one can construct a polynomial reduction from the Hamiltonian cycle problem to the satisfiability problem SAT. Given a directed graph $G = (V, E)$ with $n$ nodes, say $V = \{1, \ldots, n\}$, we need to construct in polynomial time a set $F = \tau(G)$ of clauses such that $G$ has a Hamiltonian cycle iff $\tau(G)$ is satisfiable. We need to describe a permutation of the nodes that forms a Hamiltonian cycle. For this we introduce $n^2$ boolean variables $x_{ij}$, with the intended interpretation that $x_{ij}$ is true iff node $i$ is the $j$th node in a Hamiltonian cycle.

To express that at least one node must appear as the $j$th node in a Hamiltonian cycle, we have the $n$ clauses

$$(x_{1j} \lor x_{2j} \lor \cdots \lor x_{nj}), \quad 1 \leq j \leq n. \quad (1)$$
The conjunction of these clauses is satisfied if for every \( j = 1, \ldots, n \) there is some node \( i \) which is the \( j \)th node in the cycle.

To express that only one node appears in the cycle, we have the clauses

\[
(x_{ij} \lor x_{kj}), \quad 1 \leq i, j, k \leq n, \quad i \neq k.
\] (2)

Since \((x_{ij} \lor x_{kj})\) is equivalent to \((x_{ij} \land x_{kj})\), each such clause asserts that no two distinct nodes may appear as the \( j \)th node in the cycle. Let \( S_1 \) be the set of all clauses of type (1) or (2).

The conjunction of the clauses in \( S_1 \) assert that exactly one node appear at the \( j \)th node in the Hamiltonian cycle. We still need to assert that each node \( i \) appears exactly once in the cycle. For this, we have the clauses

\[
(x_{i1} \lor x_{i2} \lor \cdots \lor x_{in}), \quad 1 \leq i \leq n,
\] (3)

and

\[
(x_{ij} \lor x_{ik}), \quad 1 \leq i, j, k \leq n, \quad j \neq k.
\] (4)

Let \( S_2 \) be the set of all clauses of type (3) or (4).

The conjunction of the clauses in \( S_1 \cup S_2 \) asserts that the \( x_{ij} \) represents a bijection of \( \{1, 2, \ldots, n\} \), in the sense that for any truth assignment \( v \) satisfying all these clauses, \( i \mapsto j \) iff \( v(x_{ij}) = T \) defines a bijection of \( \{1, 2, \ldots, n\} \).

It remains to assert that this permutation of the nodes is a Hamiltonian cycle, which means that if \( x_{ij} \) and \( x_{kj+1} \) are both true then there there must be an edge \((i, k)\). By contrapositive, this equivalent to saying that if \((i, k)\) is not an edge of \( G \), then \((x_{ij} \land x_{kj+1})\) is true, which as a clause is equivalent to \((x_{ij} \lor x_{kj})\).

Therefore, for all \((i, k)\) such that \((i, k) \notin E\) (with \( i, k \in \{1, 2, \ldots, n\} \)), we have the clauses

\[
(x_{ij} \lor x_{kj+1} \mod n), \quad j = 1, \ldots, n.
\] (5)

Let \( S_3 \) be the set of clauses of type (5). The conjunction of all the clauses in \( S_1 \cup S_2 \cup S_3 \) is the boolean formula \( F = \tau(G) \).

We leave it as an exercise to prove that \( G \) has a Hamiltonian cycle iff \( F = \tau(G) \) is satisfiable.

It is also possible to construct a reduction of the satisfiability problem to the Hamiltonian cycle problem but this is harder. It is easier to construct this reduction in two steps by introducing an intermediate problem, the exact cover problem, and to provide a polynomial reduction from the satisfiability problem to the exact cover problem, and a polynomial reduction from the exact cover problem to the Hamiltonian cycle problem. These reductions are carried out in Section 10.2.
The above construction of a set $F = \tau(G)$ of clauses from a graph $G$ asserting that $G$ has a Hamiltonian cycle iff $F$ is satisfiable illustrates the expressive power of propositional logic. Remarkably, every language in $\mathcal{NP}$ can be reduced to SAT. Thus, SAT is a hardest problem in $\mathcal{NP}$ (Since it is in $\mathcal{NP}$).

**Definition 9.6.** A language $L$ is $\mathcal{NP}$-hard if there is a polynomial reduction from every language $L_1 \in \mathcal{NP}$ to $L$. A language $L$ is $\mathcal{NP}$-complete if $L \in \mathcal{NP}$ and $L$ is $\mathcal{NP}$-hard.

Thus, an $\mathcal{NP}$-hard language is as hard to decide as any language in $\mathcal{NP}$.

The importance of $\mathcal{NP}$-complete problems stems from the following theorem:

**Theorem 9.3.** Let $L$ be an $\mathcal{NP}$-complete language. Then, $P = \mathcal{NP}$ iff $L \in P$.

Next, we prove a famous theorem of Steve Cook and Leonid Levin (proved independently): SAT is $\mathcal{NP}$-complete.

### 9.7 The Cook–Levin Theorem: SAT is $\mathcal{NP}$-Complete

Instead of showing directly that SAT is $\mathcal{NP}$-complete, which is rather complicated, we proceed in two steps, as suggested by Lewis and Papadimitriou.

1. First, we define a tiling problem adapted from H. Wang (1961) by Harry Lewis, and we prove that it is $\mathcal{NP}$-complete.
2. We show that the tiling problem can be reduced to SAT.

We are given a finite set $T = \{t_1, \ldots, t_p\}$ of tile patterns, for short, tiles. Copies of these tile patterns may be used to tile a rectangle of predetermined size $2s \times s$ ($s > 1$). However, there are constraints on the way that these tiles may be adjacent horizontally and vertically.

The **horizontal constraints** are given by a relation $H \subseteq T \times T$, and the **vertical constraints** are given by a relation $V \subseteq T \times T$.

Thus, a tiling system is a triple $T = (\mathcal{T}, V, H)$ with $V$ and $H$ as above.

The bottom row of the rectangle of tiles is specified before the tiling process begins.

For example, consider the following tile patterns:
The horizontal and the vertical constraints are that the letters on adjacent edges match (blank edges do not match).

For \( s = 3 \), given the bottom row

\[
\begin{array}{ccccccc}
  a & b & c & d & e & e & e \\
  c & c & d & d & e & e & e \\
  a & b & c & d & d & e & e \\
  a & b & c & d & d & e & e \\
  a & b & c & d & d & e & e \\
  c & c & d & d & e & e & e \\
\end{array}
\]

we have the tiling shown below:

Formally, the problem is then as follows:

**The Bounded Tiling Problem**

Given any tiling system \((T, V, H)\), any integer \( s > 1 \), and any initial row of tiles \( \sigma_0 \) (of length \( 2s \))

\[
\sigma_0: \{1, 2, \ldots, s, s+1, \ldots, 2s\} \rightarrow T,
\]

find a \( 2s \times s \)-tiling \( \sigma \) extending \( \sigma_0 \), i.e., a function

\[
\sigma: \{1, 2, \ldots, s, s+1, \ldots, 2s\} \times \{1, \ldots, s\} \rightarrow T
\]

so that
CHAPTER 9. COMPUTATIONAL COMPLEXITY; \( \mathcal{P} \) AND \( \mathcal{NP} \)

(1) \( \sigma(m, 1) = \sigma_0(m) \), for all \( m \) with \( 1 \leq m \leq 2s \).

(2) \( (\sigma(m, n), \sigma(m + 1, n)) \in H \), for all \( m \) with \( 1 \leq m \leq 2s - 1 \), and all \( n \), with \( 1 \leq n \leq s \).

(3) \( (\sigma(m, n), \sigma(m, n + 1)) \in V \), for all \( m \) with \( 1 \leq m \leq 2s \), and all \( n \), with \( 1 \leq n \leq s - 1 \).

Formally, an instance of the tiling problem is a triple \( ((T, V, H), \hat{s}, \sigma_0) \), where \( (T, V, H) \) is a tiling system, \( \hat{s} \) is the string representation of the number \( s \geq 2 \), in binary and \( \sigma_0 \) is an initial row of tiles (the bottom row).

For example, if \( s = 1025 \) (as a decimal number), then its binary representation is \( \hat{s} = 10000000001 \). The length of \( \hat{s} \) is \( \log_2 s + 1 \).

Recall that the input must be a string. This is why the number \( s \) is represented by a string in binary.

If we only included a single tile \( \sigma_0 \) in position \( (s + 1, 1) \), then the length of the input \( ((T, V, H), \hat{s}, \sigma_0) \) would be \( \log_2 s + C + 2 \) for some constant \( C \) corresponding to the length of the string encoding \( (T, V, H) \).

However, the rectangular grid has size \( 2s^2 \), which is exponential in the length \( \log_2 s + C + 2 \) of the input \( ((T, V, H), \hat{s}, \sigma_0) \). Thus, it is impossible to check in polynomial time that a proposed solution is a tiling.

However, if we include in the input the bottom row \( \sigma_0 \) of length \( 2s \), then the size of the grid is indeed polynomial in the size of the input.

**Theorem 9.4.** The tiling problem defined earlier is \( \mathcal{NP} \)-complete.

**Proof.** Let \( L \subseteq \Sigma^* \) be any language in \( \mathcal{NP} \) and let \( u \) be any string in \( \Sigma^* \). Assume that \( L \) is accepted in polynomial time bounded by \( p(|u|) \).

We show how to construct an instance of the tiling problem, \( ((T, V, H)_L, \hat{s}, \sigma_0) \), where \( s = p(|u|) + 2 \), and where the bottom row encodes the starting ID, so that \( u \in L \) iff the tiling problem \( ((T, V, H)_L, \hat{s}, \sigma_0) \) has a solution.

First, note that the problem is indeed in \( \mathcal{NP} \), since we have to guess a rectangle of size \( 2s^2 \), and that checking that a tiling is legal can indeed be done in \( O(s^2) \), where \( s \) is bounded by the the size of the input \( ((T, V, H), \hat{s}, \sigma_0) \), since the input contains the bottom row of \( 2s \) symbols (this is the reason for including the bottom row of \( 2s \) tiles in the input!).

The idea behind the definition of the tiles is that, in a solution of the tiling problem, the labels on the horizontal edges between two adjacent rows represent a legal ID, \( \text{upav} \).

In a given row, the labels on vertical edges of adjacent tiles keep track of the change of state and direction.
Let $\Gamma$ be the tape alphabet of the TM, $M$. As before, we assume that $M$ signals that it accepts $u$ by halting with the output 1 (true).

From $M$, we create the following tiles:

1. For every $a \in \Gamma$, tiles

   $a$
   $a$

2. For every $a \in \Gamma$, the bottom row uses tiles

   $a$
   $q_0, a$

   where $q_0$ is the start state.

3. For every instruction $(p, a, b, R, q) \in \delta$, for every $c \in \Gamma$, tiles

   $b$
   $q, R$
   $p, a$

   $q, c$
   $q, R$
   $c$

4. For every instruction $(p, a, b, L, q) \in \delta$, for every $c \in \Gamma$, tiles

   $q, c$
   $q, L$
   $c$

   $b$
   $q, L$
   $p, a$

5. For every halting state, $p$, tiles

   $p, 1$
   $p, 1$
The purpose of tiles of type (5) is to fill the $2s \times s$ rectangle iff $M$ accepts $u$. Since $s = p(|u|) + 2$ and the machine runs for at most $p(|u|)$ steps, the $2s \times s$ rectangle can be tiled iff $u \in L$.

The vertical and the horizontal constraints are that adjacent edges have the same label (or no label).

If $u = u_1 \cdots u_k$, the initial bottom row $\sigma_0$, of length $2s$, is:

$$
\begin{array}{cccc}
B & \ldots & q_0, u_1 & \ldots & u_k & \ldots & B
\end{array}
$$

where the tile labeled $q_0, u_1$ is in position $s + 1$.

The example below illustrates the construction:

$$
\begin{array}{cccc}
B & \ldots & B & \ldots \\
& q, c & f, 1 & \ldots \\
& q, c & f, R & \ldots \\
& q, L & 1 & \ldots \\
& c & p, a & \ldots \\
& c & p, R & \ldots \\
& r, b & a & \ldots \\
B & \ldots & B & \ldots
\end{array}
$$

We claim that $u = u_1 \cdots u_k$ is accepted by $M$ iff the tiling problem just constructed has a solution.

The upper horizontal edge of the first (bottom) row of tiles represents the starting configuration $B^*q_0uB^{s-|u|}$. By induction, we see that after $i \ (i \leq p(|u|) = s - 2)$ steps the upper horizontal edge of the $(i + 1)$th row of tiles represents the current ID $\text{upav}$ reached by the Turing machine. Since the machine runs for at most $p(|u|)$ steps and since $s = p(|u|) + 2$, when the computation stops, at most the lowest $p(|u|) + 1 = s - 1$ rows of the $2s \times s$ rectangle have been tiled. Assume the machine $M$ stops after $r \leq s - 2$ steps. Then the lowest $r + 1$ rows have been tiled, and since no further instruction can be executed (since the machine entered a halting state), the remaining $s - r - 1$ rows can be filled iff tiles of type (5) can be used iff the machine stopped in an ID containing a pair $p1$ where $p$ is a halting state. Therefore, the machine $M$ accepts $u$ iff the $2s \times s$ rectangle can be tiled.

**Remarks.**

1. The problem becomes harder if we only specify a single tile $\sigma_0$ as input, instead of a row of length $2s$. If $s$ is specified in binary (or any other base, but not in tally notation), then the $2s^2$ grid has size exponential in the length $\log_2 s + C + 2$ of the input $((\mathcal{T}, V, H), \hat{s}, \sigma_0)$, and this tiling problem is actually $\mathcal{NEXP}$-complete!
9.7. THE COOK-LEVIN THEOREM

(2) If we relax the finiteness condition and require that the entire upper half-plane be tiled, i.e., for every \( s > 1 \), there is a solution to the \( 2s \times s \)-tiling problem, then the problem is undecidable.

In 1972, Richard Karp published a list of 21 \( \mathcal{NP} \)-complete problems.

We finally prove the Cook-Levin theorem.

**Theorem 9.5.** (Cook, 1971, Levin, 1973) The satisfiability problem SAT is \( \mathcal{NP} \)-complete.

**Proof.** We reduce the tiling problem to SAT. Given a tiling problem, \( ((\mathcal{T}, V, H), \mathcal{S}, \sigma_0) \), we introduce boolean variables

\[ x_{mnt}, \]

for all \( m \) with \( 1 \leq m \leq 2s \), all \( n \) with \( 1 \leq n \leq s \), and all tiles \( t \in \mathcal{T} \).

The intuition is that \( x_{mnt} = T \) iff tile \( t \) occurs in some tiling \( \sigma \) so that \( \sigma(m,n) = t \).

We define the following clauses:

1. For all \( m, n \) in the correct range, as above,
   \[ (x_{mnt_1} \lor x_{mnt_2} \lor \cdots \lor x_{mnt_p}), \]
   for all \( p \) tiles in \( \mathcal{T} \).
   This clause states that every position in \( \sigma \) is tiled.

2. For any two distinct tiles \( t \neq t' \in \mathcal{T} \), for all \( m, n \) in the correct range, as above,
   \[ (\overline{x_{mnt}} \lor \overline{x_{mnt'}}), \]
   This clause states that a position may not be occupied by more than one tile.

3. For every pair of tiles \( (t, t') \in \mathcal{T} \times \mathcal{T} - H \), for all \( m \) with \( 1 \leq m \leq 2s - 1 \), and all \( n \), with \( 1 \leq n \leq s \),
   \[ (\overline{x_{mnt}} \lor \overline{x_{m+1nt'}}), \]
   This clause enforces the horizontal adjacency constraints.

4. For every pair of tiles \( (t, t') \in \mathcal{T} \times \mathcal{T} - V \), for all \( m \) with \( 1 \leq m \leq 2s \), and all \( n \), with \( 1 \leq n \leq s - 1 \),
   \[ (\overline{x_{mnt}} \lor \overline{x_{mn+1t'}}), \]
   This clause enforces the vertical adjacency constraints.
(5) For all $m$ with $1 \leq m \leq 2s$,
\[ (x_{m1\sigma_0(m)}). \]

This clause states that the bottom row is correctly tiled with $\sigma_0$.

It is easily checked that the tiling problem has a solution iff the conjunction of the clauses just defined is satisfiable. Thus, SAT is $\mathcal{NP}$-complete.

We sharpen Theorem 9.5 to prove that 3-SAT is also $\mathcal{NP}$-complete. This is the satisfiability problem for clauses containing at most three literals.

We know that we can’t go further and retain $\mathcal{NP}$-completeness, since 2-SAT is in $\mathcal{P}$.

**Theorem 9.6.** (Cook, 1971) *The satisfiability problem 3-SAT is $\mathcal{NP}$-complete.*

**Proof.** We have to break “long clauses”
\[ C = (L_1 \lor \cdots \lor L_k), \]
i.e., clauses containing $k \geq 4$ literals, into clauses with at most three literals, in such a way that satisfiability is preserved.

For example, consider the following clause with $k = 6$ literals:
\[ C = (L_1 \lor L_2 \lor L_3 \lor L_4 \lor L_5 \lor L_6). \]

We create 3 new boolean variables $y_1, y_2, y_3$, and the 4 clauses
\[ (L_1 \lor L_2 \lor y_1), (\overline{y_1} \lor L_3 \lor y_2), (\overline{y_2} \lor L_4 \lor y_3), (\overline{y_3} \lor L_5 \lor L_6). \]

Let $C'$ be the conjunction of these clauses. We claim that $C$ is satisfiable iff $C'$ is.

Assume that $C'$ is satisfiable but $C$ is not. If so, in any truth assignment $v$, $v(L_i) = F$, for $i = 1, 2, \ldots, 6$. To satisfy the first clause, we must have $v(y_1) = T$. Then to satisfy the second clause, we must have $v(y_2) = T$, and similarly satisfy the third clause, we must have $v(y_3) = T$. However, since $v(L_5) = F$ and $v(L_6) = F$, the only way to satisfy the fourth clause is to have $v(y_3) = F$, contradicting that $v(y_3) = T$. Thus, $C$ is indeed satisfiable.

Let us now assume that $C$ is satisfiable. This means that there is a smallest index $i$ such that $L_i$ is satisfied.

Say $i = 1$, so $v(L_1) = T$. Then if we let $v(y_1) = v(y_2) = v(y_3) = F$, we see that $C'$ is satisfied.

Say $i = 2$, so $v(L_1) = F$ and $v(L_2) = T$. Again if we let $v(y_1) = v(y_2) = v(y_3) = F$, we see that $C'$ is satisfied.
Say $i = 3$, so $v(L_1) = \text{F}$, $v(L_2) = \text{F}$, and $v(L_3) = \text{T}$. If we let $v(y_1) = \text{T}$ and $v(y_2) = v(y_3) = \text{F}$, we see that $C'$ is satisfied.

Say $i = 4$, so $v(L_1) = \text{F}$, $v(L_2) = \text{F}$, $v(L_3) = \text{F}$, and $v(L_4) = \text{T}$. If we let $v(y_1) = \text{T}$, $v(y_2) = \text{T}$ and $v(y_3) = \text{F}$, we see that $C'$ is satisfied.

Say $i = 5$, so $v(L_1) = \text{F}$, $v(L_2) = \text{F}$, $v(L_3) = \text{F}$, $v(L_4) = \text{F}$, and $v(L_5) = \text{T}$. If we let $v(y_1) = \text{T}$, $v(y_2) = \text{T}$ and $v(y_3) = \text{T}$, we see that $C'$ is satisfied.

Say $i = 6$, so $v(L_1) = \text{F}$, $v(L_2) = \text{F}$, $v(L_3) = \text{F}$, $v(L_4) = \text{F}$, $v(L_5) = \text{F}$, and $v(L_6) = \text{T}$. Again, if we let $v(y_1) = \text{T}$, $v(y_2) = \text{T}$ and $v(y_3) = \text{T}$, we see that $C'$ is satisfied.

Therefore if $C$ is satisfied, then $C'$ is satisfied in all cases.

In general, for every long clause, create $k - 3$ new boolean variables $y_1, \ldots y_{k-3}$, and the $k - 2$ clauses

$$(L_1 \lor L_2 \lor y_1), (\overline{y}_1 \lor L_3 \lor y_2), (\overline{y}_2 \lor L_4 \lor y_3), \ldots,$$

$$(\overline{y}_{k-3} \lor L_{k-2} \lor y_{k-3}), (\overline{y}_{k-3} \lor L_{k-1} \lor L_k).$$

Let $C'$ be the conjunction of these clauses. We claim that $C$ is satisfiable iff $C'$ is.

Assume that $C'$ is satisfiable, but that $C$ is not. Then, for every truth assignment $v$, we have $v(L_i) = \text{F}$, for $i = 1, \ldots, k$.

However, $C'$ is satisfied by some $v$, and the only way this can happen is that $v(y_1) = \text{T}$, to satisfy the first clause. Then, $v(\overline{y}_1) = \text{F}$, and we must have $v(y_2) = \text{T}$, to satisfy the second clause.

By induction, we must have $v(y_{k-3}) = \text{T}$, to satisfy the next to the last clause. However, the last clause is now false, a contradiction.

Thus, if $C'$ is satisfiable, then so is $C$.

Conversely, assume that $C$ is satisfiable. If so, there is some truth assignment, $v$, so that $v(C) = \text{T}$, and thus, there is a smallest index $i$, with $1 \leq i \leq k$, so that $v(L_i) = \text{T}$ (and so, $v(L_j) = \text{F}$ for all $j < i$).

Let $v'$ be the assignment extending $v$ defined so that

$$v'(y_j) = \text{F} \quad \text{if} \quad \max\{1, i-1\} \leq j \leq k - 3,$$

and $v'(y_j) = \text{T}$, otherwise.

It is easily checked that $v'(C') = \text{T}$. \hfill \Box

Another version of 3-SAT can be considered, in which every clause has exactly three literals. We will call this the problem exact 3-SAT.

**Theorem 9.7.** (Cook, 1971) *The satisfiability problem for exact 3-SAT is \text{NP}-complete.*
Proof. A clause of the form \((L)\) is satisfiable iff the following four clauses are satisfiable:
\[
(L \lor u \lor v), (L \lor \overline{u} \lor v), (L \lor u \lor \overline{v}), (L \lor \overline{u} \lor \overline{v}).
\]

A clause of the form \((L_1 \lor L_2)\) is satisfiable iff the following two clauses are satisfiable:
\[
(L_1 \lor L_2 \lor u), (L_1 \lor L_2 \lor \overline{u}).
\]

Thus, we have a reduction of 3-SAT to exact 3-SAT.

We now make some remarks on the conversion of propositions to CNF.

Recall that the set of propositions (over the connectives \(\lor\), \(\land\), and \(\neg\)) is defined inductively as follows:

1. Every propositional letter, \(x \in \mathbf{PV}\), is a proposition (an atomic proposition).
2. If \(A\) is a proposition, then \(\neg A\) is a proposition.
3. If \(A\) and \(B\) are propositions, then \((A \lor B)\) is a proposition.
4. If \(A\) and \(B\) are propositions, then \((A \land B)\) is a proposition.

Two propositions \(A\) and \(B\) are equivalent, denoted \(A \equiv B\), if

\[
v \models A \text{ iff } v \models B
\]

for all truth assignments, \(v\).

It is easy to show that \(A \equiv B\) iff the proposition
\[
(\neg A \lor B) \land (\neg B \lor A)
\]
is valid.

Every proposition, \(A\), is equivalent to a proposition, \(A'\), in CNF.

There are several ways of proving this fact. One method is algebraic, and consists in using the algebraic laws of boolean algebra.

First, one may convert a proposition to negation normal form, or nnf. A proposition is in nnf if occurrences of \(\neg\) only appear in front of propositional variables, but not in front of compound propositions.

Any proposition can be converted to an equivalent one in nnf by using the de Morgan laws:
\[
\neg(A \lor B) \equiv (\neg A \land \neg B)
\]
\[
\neg(A \land B) \equiv (\neg A \lor \neg B)
\]
\[
\neg\neg A \equiv A.
\]
Then, a proposition in \( \text{nnf} \) can be converted to \( \text{CNF} \), but the question of uniqueness of the \( \text{CNF} \) is a bit tricky.

For example, the proposition
\[
A = (u \land (x \lor y)) \lor (\neg u \land (x \lor y))
\]
has
\[
\begin{align*}
A_1 &= (u \lor x \lor y) \land (\neg u \lor x \lor y) \\
A_2 &= (u \lor \neg u) \land (x \lor y) \\
A_3 &= x \lor y,
\end{align*}
\]
as equivalent propositions in \( \text{CNF} \! \)!

We can get a unique \( \text{CNF} \) equivalent to a given proposition if we do the following:

1. Let \( \text{Var}(A) = \{x_1, \ldots, x_m\} \) be the set of variables occurring in \( A \).
2. Define a maxterm w.r.t. \( \text{Var}(A) \) as any disjunction of \( m \) pairwise distinct literals formed from \( \text{Var}(A) \), and not containing both some variable \( x_i \) and its negation \( \neg x_i \).
3. Then, it can be shown that for any proposition \( A \) that is not a tautology, there is a unique proposition in \( \text{CNF} \) equivalent to \( A \), whose clauses consist of maxterms formed from \( \text{Var}(A) \).

The above definition can yield strange results. For instance, the \( \text{CNF} \) of any unsatisfiable proposition with \( m \) distinct variables is the conjunction of all of its \( 2^m \) maxterms!

The above notion does not cope well with minimality.

For example, according to the above, the \( \text{CNF} \) of
\[
A = (u \land (x \lor y)) \lor (\neg u \land (x \lor y))
\]
should be
\[
A_1 = (u \lor x \lor y) \land (\neg u \lor x \lor y).
\]

There are also propositions such that any equivalent proposition in \( \text{CNF} \) has size exponential in terms of the original proposition.

Here is such an example:
\[
A = (x_1 \land x_2) \lor (x_3 \land x_4) \lor \cdots \lor (x_{2^n-1} \land x_{2n}).
\]

Observe that it is in DNF.

We will prove a little later that any \( \text{CNF} \) for \( A \) contains \( 2^n \) occurrences of variables.
A nice method to convert a proposition in nnf to CNF is to construct a tree whose nodes are labeled with sets of propositions using the following (Gentzen-style) rules:

\[
P, \Delta \quad Q, \Delta \\
\frac{(P \land Q), \Delta}{(P \lor Q), \Delta}
\]

and

\[
P, Q, \Delta \\
\frac{(P \lor Q), \Delta}{(P \land Q), \Delta}
\]

where \(\Delta\) stands for any set of propositions (even empty), and the comma stands for union. Thus, it is assumed that \((P \land Q) \notin \Delta\) in the first case, and that \((P \lor Q) \notin \Delta\) in the second case.

Since we interpret a set, \(\Gamma\), of propositions as a disjunction, a valuation, \(v\), satisfies \(\Gamma\) iff it satisfies some proposition in \(\Gamma\).

Observe that a valuation \(v\) satisfies the conclusion of a rule iff it satisfies both premises in the first case, and the single premise in the second case.

Using these rules, we can build a finite tree whose leaves are labeled with sets of literals.

By the above observation, a valuation \(v\) satisfies the proposition labeling the root of the tree iff it satisfies all the propositions labeling the leaves of the tree.

But then, a CNF for the original proposition \(A\) (in nnf, at the root of the tree) is the conjunction of the clauses appearing as the leaves of the tree.

We may exclude the clauses that are tautologies, and we may discover in the process that \(A\) is a tautology (when all leaves are tautologies).

Going back to our “bad” proposition, \(A\), by induction, we see that any tree for \(A\) has \(2^n\) leaves.

However, it should be noted that for any proposition, \(A\), we can construct in polynomial time a formula, \(A'\), in CNF, so that \(A\) is satisfiable iff \(A'\) is satisfiable, by creating new variables.

We proceed recursively. The trick is that we replace

\[
(C_1 \land \cdots \land C_m) \lor (D_1 \land \cdots \land D_n)
\]

by

\[
(C_1 \lor y) \land \cdots \land (C_m \lor y) \land (D_1 \lor \overline{y}) \land \cdots \land (D_n \lor \overline{y}),
\]

where the \(C_i\)'s and the \(D_j\)'s are clauses, and \(y\) is a new variable.

It can be shown that the number of new variables required is at most quadratic in the size of \(A\).

Warning: In general, the proposition \(A'\) is not equivalent to the proposition \(A\).
Rules for dealing for \( \neg \) can also be created. In this case, we work with pairs of sets of propositions,

\[ \Gamma \rightarrow \Delta, \]

where, the propositions in \( \Gamma \) are interpreted conjunctively, and the propositions in \( \Delta \) are interpreted disjunctively.

We obtain a sound and complete proof system for propositional logic (a “Gentzen-style” proof system, see Gallier’s *Logic for Computer Science*).
Chapter 10

Some $\mathcal{NP}$-Complete Problems

10.1 Statements of the Problems

In this chapter we will show that certain classical algorithmic problems are $\mathcal{NP}$-complete. This chapter is heavily inspired by Lewis and Papadimitriou’s excellent treatment [1]. In order to study the complexity of these problems in terms of resource (time or space) bounded Turing machines (or RAM programs), it is crucial to be able to encode instances of a problem $P$ as strings in a language $L_P$. Then an instance of a problem $P$ is solvable iff the corresponding string belongs to the language $L_P$. This implies that our problems must have a yes–no answer, which is not always the usual formulation of optimization problems where what is required is to find some optimal solution, that is, a solution minimizing or maximizing some objective (cost) function $F$. For example the standard formulation of the traveling salesman problem asks for a tour (of the cities) of minimal cost.

Fortunately, there is a trick to reformulate an optimization problem as a yes–no answer problem, which is to explicitly incorporate a budget (or cost) term $B$ into the problem, and instead of asking whether some objective function $F$ has a minimum or a maximum $w$, we ask whether there is a solution $w$ such that $F(w) \leq B$ in the case of a minimum solution, or $F(w) \geq B$ in the case of a maximum solution.

If we are looking for a minimum of $F$, we try to guess the minimum value $B$ of $F$ and then we solve the problem of finding $w$ such that $F(w) \leq B$. If our guess for $B$ is too small, then we fail. In this case, we try again with a larger value of $B$. Otherwise, if $B$ was not too small we find some $w$ such that $F(w) \leq B$, but $w$ may not correspond to a minimum of $F$, so we try again with a smaller value of $B$, and so on. This yields an approximation method to find a minimum of $F$.

Similarly, if we are looking for a maximum of $F$, we try to guess the maximum value $B$ of $F$ and then we solve the problem of finding $w$ such that $F(w) \geq B$. If our guess for $B$ is too large, then we fail. In this case, we try again with a smaller value of $B$. Otherwise, if $B$ was not too large we find some $w$ such that $F(w) \geq B$, but $w$ may not correspond to a maximum of $F$, so we try again with a greater value of $B$, and so on. This yields an
approximation method to find a maximum of $F$.

We will see several examples of this technique in Problems 5–8 listed below.

The problems that will consider are

1. **Exact Cover**
2. Hamiltonian Cycle for directed graphs
3. Hamiltonian Cycle for undirected graphs
4. The Traveling Salesman Problem
5. Independent Set
6. Clique
7. Node Cover
8. Knapsack, also called subset sum
9. Inequivalence of $*$-free Regular Expressions
10. The 0-1-integer programming problem

We begin by describing each of these problems.

1. **Exact Cover**

We are given a finite nonempty set $U = \{u_1, \ldots, u_n\}$ (the universe), and a family $\mathcal{F} = \{S_1, \ldots, S_m\}$ of $m \geq 1$ nonempty subsets of $U$. The question is whether there is an exact cover, that is, a subfamily $\mathcal{C} \subseteq \mathcal{F}$ of subsets in $\mathcal{F}$ such that the sets in $\mathcal{C}$ are disjoint and their union is equal to $U$.

For example, let $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$, and let $\mathcal{F}$ be the family

$$\mathcal{F} = \{\{u_1, u_3\}, \{u_2, u_3, u_6\}, \{u_1, u_5\}, \{u_2, u_3, u_4\}, \{u_5, u_6\}, \{u_2, u_4\}\}.$$

The subfamily

$$\mathcal{C} = \{\{u_1, u_3\}, \{u_5, u_6\}, \{u_2, u_4\}\}$$

is an exact cover.

It is easy to see that **Exact Cover** is in \(\mathcal{NP}\). To prove that it is \(\mathcal{NP}\)-complete, we will reduce the **Satisfiability Problem** to it. This means that we provide a method running in polynomial time that converts every instance of the **Satisfiability Problem** to an instance of **Exact Cover**, such that the first problem has a solution iff the converted problem has a solution.
(2) **Hamiltonian Cycle (for Directed Graphs)**

Recall that a directed graph $G$ is a pair $G = (V, E)$, where $E \subseteq V \times V$. Elements of $V$ are called nodes (or vertices). A pair $(u, v) \in E$ is called an edge of $G$. We will restrict ourselves to simple graphs, that is, graphs without edges of the form $(u, u)$; equivalently, $G = (V, E)$ is a simple graph if whenever $(u, v) \in E$, then $u \neq v$.

Given any two nodes $u, v \in V$, a path from $u$ to $v$ is any sequence of $n+1$ edges $(n \geq 0)$

$$(u, v_1), (v_1, v_2), \ldots, (v_n, v).$$

(If $n = 0$, a path from $u$ to $v$ is simply a single edge, $(u, v)$.)

A directed graph $G$ is strongly connected if for every pair $(u, v) \in V \times V$, there is a path from $u$ to $v$. A closed path, or cycle, is a path from some node $u$ to itself. We will restrict our attention to finite graphs, i.e. graphs $(V, E)$ where $V$ is a finite set.

**Definition 10.1.** Given a directed graph $G$, a Hamiltonian cycle is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

**Hamiltonian Cycle Problem (for Directed Graphs):** Given a directed graph $G$, is there an Hamiltonian cycle in $G$?

Is there is a Hamiltonian cycle in the directed graph $D$ shown in Figure 10.1?

![Figure 10.1: A tour “around the world.”](image-url)
Finding a Hamiltonian cycle in this graph does not appear to be so easy! A solution is shown in Figure 10.2 below.

![Hamiltonian cycle diagram]

Figure 10.2: A Hamiltonian cycle in $D$.

It is easy to see that **Hamiltonian Cycle (for Directed Graphs)** is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact Cover** to an instance of **Hamiltonian Cycle (for Directed Graphs)** such that the first problem has a solution if the converted problem has a solution. This is perhaps the hardest reduction.

(3) **Hamiltonian Cycle (for Undirected Graphs)**

Recall that an **undirected graph** $G$ is a pair $G = (V, E)$, where $E$ is a set of subsets $\{u, v\}$ of $V$ consisting of exactly two distinct elements. Elements of $V$ are called **nodes** (or **vertices**). A pair $\{u, v\} \in E$ is called an **edge** of $G$.

Given any two nodes $u, v \in V$, a **path from $u$ to $v$** is any sequence of $n$ nodes ($n \geq 2$)

$$u = u_1, u_2, \ldots, u_n = v$$

such that $\{u_i, u_{i+1}\} \in E$ for $i = 1, \ldots, n - 1$. (If $n = 2$, a path from $u$ to $v$ is simply a single edge, $\{u, v\}$.)

An undirected graph $G$ is **connected** if for every pair $(u, v) \in V \times V$, there is a path from $u$ to $v$. A **closed path**, or **cycle**, is a path from some node $u$ to itself.
Definition 10.2. Given an undirected graph $G$, a Hamiltonian cycle is a cycle that passes through all the nodes exactly once (note, some edges may not be traversed at all).

Hamiltonian Cycle Problem (for Undirected Graphs): Given an undirected graph $G$, is there an Hamiltonian cycle in $G$?

An instance of this problem is obtained by changing every directed edge in the directed graph of Figure 10.1 to an undirected edge. The directed Hamiltonian cycle given in Figure 10.2 is also an undirected Hamiltonian cycle of the undirected graph of Figure 10.3.

![Figure 10.3: A tour “around the world,” undirected version.](image)

We see immediately that Hamiltonian Cycle (for Undirected Graphs) is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete, we will reduce Hamiltonian Cycle (for Directed Graphs) to it. This means that we provide a method running in polynomial time that converts every instance of Hamiltonian Cycle (for Directed Graphs) to an instance of Hamiltonian Cycle (for Undirected Graphs) such that the first problem has a solution iff the converted problem has a solution. This is an easy reduction.

(4) Traveling Salesman Problem
We are given a set \( \{c_1, c_2, \ldots, c_n\} \) of \( n \geq 2 \) cities, and an \( n \times n \) matrix \( D = (d_{ij}) \) of nonnegative integers, where \( d_{ij} \) is the distance (or cost) of traveling from city \( c_i \) to city \( c_j \). We assume that \( d_{ii} = 0 \) and \( d_{ij} = d_{ji} \) for all \( i, j \), so that the matrix \( D \) is symmetric and has zero diagonal.

**Traveling Salesman Problem**: Given some \( n \times n \) matrix \( D = (d_{ij}) \) as above and some integer \( B \geq 0 \) (the budget of the traveling salesman), find a permutation \( \pi \) of \( \{1, 2, \ldots, n\} \) such that

\[
c(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)} \leq B.
\]

The quantity \( c(\pi) \) is the cost of the trip specified by \( \pi \). The Traveling Salesman Problem has been stated in terms of a budget so that it has a yes or no answer, which allows us to convert it into a language. A minimal solution corresponds to the smallest feasible value of \( B \).

**Example 10.1.** Consider the 4 \( \times \) 4 symmetric matrix given by

\[
D = \begin{pmatrix}
0 & 2 & 1 & 1 \\
2 & 0 & 1 & 1 \\
1 & 1 & 0 & 3 \\
1 & 1 & 3 & 0
\end{pmatrix},
\]

and the budget \( B = 4 \). The tour specified by the permutation

\[
\pi = \begin{pmatrix}
1 & 2 & 3 & 4 \\
1 & 4 & 2 & 3
\end{pmatrix}
\]

has cost 4, since

\[
c(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + d_{\pi(3)\pi(4)} + d_{\pi(4)\pi(1)}
\]

\[
= d_{14} + d_{42} + d_{23} + d_{31}
\]

\[
= 1 + 1 + 1 + 1 = 4.
\]

The cities in this tour are traversed in the order

\[
(1, 4, 2, 3, 1).
\]

It is clear that the **Traveling Salesman Problem** is in \( \mathcal{NP} \). To show that it is \( \mathcal{NP} \)-complete, we reduce the **Hamiltonian Cycle Problem (Undirected Graphs)** to it. This means that we provide a method running in polynomial time that converts every instance of **Hamiltonian Cycle Problem (Undirected Graphs)** to an instance of the **Traveling Salesman Problem** such that the first problem has a solution iff the converted problem has a solution.
(5) **Independent Set**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set $C$ of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is no edge \{v_i, v_j\} $\in E$?

A maximal independent set with 3 nodes is shown in Figure 10.4. A maximal solution corresponds to the largest feasible value of $K$. The problem **Independent Set** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Exact 3-Satisfiability** to it. This means that we provide a method running in polynomial time that converts every instance of **Exact 3-Satisfiability** to an instance of **Independent Set** such that the first problem has a solution iff the converted problem has a solution.

(6) **Clique**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $K \geq 2$, is there a set $C$ of nodes with $|C| \geq K$ such that for all $v_i, v_j \in C$, there is some edge \{v_i, v_j\} $\in E$? Equivalently, does $G$ contain a complete subgraph with at least $K$ nodes?

A maximal clique with 4 nodes is shown in Figure 10.5. A maximal solution corresponds to the largest feasible value of $K$. The problem **Clique** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it. This means that we provide a method running in polynomial time that converts every instance of **Independent
Set to an instance of Clique such that the first problem has a solution iff the converted problem has a solution.

(7) **Node Cover**

The problem is this: Given an undirected graph $G = (V, E)$ and an integer $B \geq 2$, is there a set $C$ of nodes with $|C| \leq B$ such that $C$ covers all edges in $G$, which means that for every edge $\{v_i, v_j\} \in E$, either $v_i \in C$ or $v_j \in C$?

A minimal node cover with 6 nodes is shown in Figure 10.6. A minimal solution corresponds to the smallest feasible value of $B$. The problem **Node Cover** is obviously in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce **Independent Set** to it. This means that we provide a method running in polynomial time that converts every instance of **Independent Set** to an instance of **Node Cover** such that the first problem has a solution iff the converted problem has a solution.

The Node Cover problem has the following interesting interpretation: think of the nodes of the graph as rooms of a museum (or art gallery etc.), and each edge as a straight corridor that joins two rooms. Then Node Cover may be useful in assigning as few as possible guards to the rooms, so that all corridors can be seen by a guard.

(8) **Knapsack (also called Subset sum)**

The problem is this: Given a finite nonempty set $S = \{a_1, a_2, \ldots, a_n\}$ of nonnegative integers, and some integer $K \geq 0$, all represented in binary, is there a nonempty subset
10.1. Statements of the Problems

Figure 10.6: A minimal Node Cover in a graph

$I \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in I} a_i = K?$$

A “concrete” realization of this problem is that of a hiker who is trying to fill her/his backpack to its maximum capacity with items of varying weights or values.

It is easy to see that the Knapsack Problem is in \(\mathcal{NP}\). To show that it is \(\mathcal{NP}\)-complete, we reduce Exact Cover to it. This means that we provide a method running in polynomial time that converts every instance of Exact Cover to an instance of Knapsack Problem such that the first problem has a solution iff the converted problem has a solution.

Remark: The 0-1 Knapsack Problem is defined as the following problem. Given a set of \(n\) items, numbered from 1 to \(n\), each with a weight \(w_i \in \mathbb{N}\) and a value \(v_i \in \mathbb{N}\), given a maximum capacity \(W \in \mathbb{N}\) and a budget \(B \in \mathbb{N}\), is there a set of \(n\) variables \(x_1, \ldots, x_n\) with \(x_i \in \{0, 1\}\) such that

$$\sum_{i=1}^{n} x_i v_i \geq B,$$

$$\sum_{i=1}^{n} x_i w_i \leq W.$$
Informally, the problem is to pick items to include in the knapsack so that the sum of the values exceeds a given minimum $B$ (the goal is to maximize this sum), and the sum of the weights is less than or equal to the capacity $W$ of the knapsack. A maximal solution corresponds to the largest feasible value of $B$.

The Knapsack Problem as we defined it (which is how Lewis and Papadimitriou define it) is the special case where $v_i = w_i = 1$ for $i = 1, \ldots, n$ and $W = B$. For this reason, it is also called the Subset Sum Problem. Clearly, the Knapsack (Subset Sum) Problem reduces to the 0-1 Knapsack Problem, and thus the 0-1 Knapsack Problem is also NP-complete.

(9) Inequivalence of ∗-free Regular Expressions

Recall that the problem of deciding the equivalence $R_1 \cong R_2$ of two regular expressions $R_1$ and $R_2$ is the problem of deciding whether $R_1$ and $R_2$ define the same language, that is, $\mathcal{L}[R_1] = \mathcal{L}[R_2]$. Is this problem in $\mathcal{NP}$?

In order to show that the equivalence problem for regular expressions is in $\mathcal{NP}$ we would have to be able to somehow check in polynomial time that two expressions define the same language, but this is still an open problem.

What might be easier is to decide whether two regular expressions $R_1$ and $R_2$ are inequivalent. For this, we just have to find a string $w$ such that either $w \in \mathcal{L}[R_1] - \mathcal{L}[R_2]$ or $w \in \mathcal{L}[R_2] - \mathcal{L}[R_1]$. The problem is that if we can guess such a string $w$, we still have to check in polynomial time that $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, and this implies that there is a bound on the length of $w$ which is polynomial in the sizes of $R_1$ and $R_2$. Again, this is an open problem.

To obtain a problem in $\mathcal{NP}$ we have to consider a restricted type of regular expressions, and it turns out that ∗-free regular expressions are the right candidate. A ∗-free regular expression is a regular expression which is built up from the atomic expressions using only + and ·, but not ∗. For example,

$$R = ((a + b)aa(a + b) + aba(a + b)b)$$

is such an expression.

It is easy to see that if $R$ is a ∗-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$. In particular, $\mathcal{L}[R]$ is finite. The above observation shows that if $R_1$ and $R_2$ are ∗-free and if there is a string $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, then $|w| \leq |R_1| + |R_2|$, so we can indeed check this in polynomial time. It follows that the inequivalence problem for ∗-free regular expressions is in $\mathcal{NP}$. To show that it is $\mathcal{NP}$-complete, we reduce the Satisfiability Problem to it. This means that we provide a method running in polynomial time that converts every instance of Satisfiability
Problem to an instance of Inequivalence of Regular Expressions such that the first problem has a solution iff the converted problem has a solution.

Observe that both problems of Inequivalence of Regular Expressions and Equivalence of Regular Expressions are as hard as Inequivalence of $*$-free Regular Expressions, since if we could solve the first two problems in polynomial time, then we could solve Inequivalence of $*$-free Regular Expressions in polynomial time, but since this problem is $\mathcal{NP}$-complete, we would have $\mathcal{P} = \mathcal{NP}$. This is very unlikely, so the complexity of Equivalence of Regular Expressions remains open.

(10) 0-1 integer programming problem

Let $A$ be any $p \times q$ matrix with integer coefficients and let $b \in \mathbb{Z}^p$ be any vector with integer coefficients. The 0-1 integer programming problem is to find whether a system of $p$ linear equations in $q$ variables

\[
\begin{align*}
    a_{11}x_1 + \cdots + a_{1q}x_q &= b_1 \\
    &\vdots \\
    a_{i1}x_1 + \cdots + a_{iq}x_q &= b_i \\
    &\vdots \\
    a_{p1}x_1 + \cdots + a_{pq}x_q &= b_p
\end{align*}
\]

with $a_{ij}, b_i \in \mathbb{Z}$ has any solution $x \in \{0, 1\}^q$, that is, with $x_i \in \{0, 1\}$. In matrix form, if we let

\[
A = \begin{pmatrix} a_{11} & \cdots & a_{1q} \\ \vdots & \ddots & \vdots \\ a_{p1} & \cdots & a_{pq} \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_p \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_q \end{pmatrix},
\]

then we write the above system as

\[
Ax = b.
\]

It is immediate that 0-1 integer programming problem is in $\mathcal{NP}$. To prove that it is $\mathcal{NP}$-complete we reduce the bounded tiling problem to it. This means that we provide a method running in polynomial time that converts every instance of the bounded tiling problem to an instance of the 0-1 integer programming problem such that the first problem has a solution iff the converted problem has a solution.

10.2 Proofs of $\mathcal{NP}$-Completeness

(1) Exact Cover
To prove that **Exact Cover** is \( \mathcal{NP} \)-complete, we reduce the **Satisfiability Problem** to it:

**Satisfiability Problem \( \leq_P \) Exact Cover**

Given a set \( F = \{ C_1, \ldots, C_\ell \} \) of \( \ell \) clauses constructed from \( n \) propositional variables \( x_1, \ldots, x_n \), we must construct in polynomial time an instance \( \tau(F) = (U, \mathcal{F}) \) of **Exact Cover** such that \( F \) is satisfiable iff \( \tau(F) \) has a solution.

**Example 10.2.** If

\[
F = \{ C_1 = (x_1 \lor \overline{x_2}), \ C_2 = (\overline{x_1} \lor x_2 \lor x_3), \ C_3 = (x_2), \ C_4 = (\overline{x_2} \lor \overline{x_3}) \},
\]

then the universe \( U \) is given by

\[
U = \{ x_1, x_2, x_3, C_1, C_2, C_3, C_4, p_{11}, p_{12}, p_{21}, p_{22}, p_{23}, p_{31}, p_{41}, p_{42} \},
\]

and the family \( \mathcal{F} \) consists of the subsets

- \( T_{1,F} = \{ x_1, p_{11} \} \)
- \( T_{1,T} = \{ x_1, p_{21} \} \)
- \( T_{2,F} = \{ x_2, p_{22}, p_{31} \} \)
- \( T_{2,T} = \{ x_2, p_{12}, p_{41} \} \)
- \( T_{3,F} = \{ x_3, p_{23} \} \)
- \( T_{3,T} = \{ x_3, p_{42} \} \)
- \( \{ C_1, p_{11} \}, \{ C_1, p_{12} \}, \{ C_2, p_{21} \}, \{ C_2, p_{22} \}, \{ C_2, p_{23} \}, \{ C_3, p_{31} \}, \{ C_4, p_{41} \}, \{ C_4, p_{42} \}. \)

It is easy to check that the set \( \mathcal{C} \) consisting of the following subsets is an exact cover:

- \( T_{1,T} = \{ x_1, p_{21} \}, T_{2,T} = \{ x_2, p_{12}, p_{41} \}, T_{3,F} = \{ x_3, p_{23} \}, \)
- \( \{ C_1, p_{11} \}, \{ C_2, p_{22} \}, \{ C_3, p_{31} \}, \{ C_4, p_{42} \}. \)

The general method to construct \( (U, \mathcal{F}) \) from \( F = \{ C_1, \ldots, C_\ell \} \) proceeds as follows. Say

\[
C_j = (L_{j1} \lor \cdots \lor L_{jm_j})
\]

is the \( j \)th clause in \( F \), where \( L_{jk} \) denotes the \( k \)th literal in \( C_j \) and \( m_j \geq 1 \). The universe of \( \tau(F) \) is the set

\[
U = \{ x_i \mid 1 \leq i \leq n \} \cup \{ C_j \mid 1 \leq j \leq \ell \} \cup \{ p_{jk} \mid 1 \leq j \leq \ell, \ 1 \leq k \leq m_j \}
\]

where in the third set \( p_{jk} \) corresponds to the \( k \)th literal in \( C_j \).

The following subsets are included in \( \mathcal{F} \):
10.2. PROOFS OF $NP$-COMPLETENESS

(a) There is a set $\{p_{jk}\}$ for every $p_{jk}$.

(b) For every boolean variable $x_i$, the following two sets are in $\mathcal{F}$:

\[ T_{i,\mathbf{T}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = \overline{x_i}\} \]

which contains $x_i$ and all negative occurrences of $x_i$, and

\[ T_{i,\mathbf{F}} = \{x_i\} \cup \{p_{jk} \mid L_{jk} = x_i\} \]

which contains $x_i$ and all its positive occurrences. Note carefully that $T_{i,\mathbf{T}}$ involves negative occurrences of $x_i$ whereas $T_{i,\mathbf{F}}$ involves positive occurrences of $x_i$.

(c) For every clause $C_j$, the $m_j$ sets $\{C_j, p_{jk}\}$ are in $\mathcal{F}$.

It remains to prove that $F$ is satisfiable iff $\tau(F)$ has a solution. We claim that if $v$ is a truth assignment that satisfies $F$, then we can make an exact cover $\mathcal{C}$ as follows:

For each $x_i$, we put the subset $T_{i,\mathbf{T}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{T}$, else we we put the subset $T_{i,\mathbf{F}}$ in $\mathcal{C}$ iff $v(x_i) = \mathbf{F}$. Also, for every clause $C_j$, we put some subset $\{C_j, p_{jk}\}$ in $\mathcal{C}$ for a literal $L_{jk}$ which is made true by $v$. By construction of $T_{i,\mathbf{T}}$ and $T_{i,\mathbf{F}}$, this $p_{jk}$ is not in any set in $\mathcal{C}$ selected so far. Since by hypothesis $F$ is satisfiable, such a literal exists for every clause. Having covered all $x_i$ and $C_j$, we put a set $\{p_{jk}\}$ in $\mathcal{C}$ for every remaining $p_{jk}$ which has not yet been covered by the sets already in $\mathcal{C}$.

Going back to Example 10.2, the truth assignment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$ satisfies $F$, so we put

\[
\begin{align*}
T_{1,\mathbf{T}} &= \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\}, \\
\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}
\end{align*}
\]

in $\mathcal{C}$.

We leave as an exercise to check that the above procedure works.

Conversely, if $\mathcal{C}$ is an exact cover of $\tau(F)$, we define a truth assignment as follows:

For every $x_i$, if $T_{i,\mathbf{T}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{T}$, else if $T_{i,\mathbf{F}}$ is in $\mathcal{C}$, then we set $v(x_i) = \mathbf{F}$. We leave it as an exercise to check that this procedure works.

**Example 10.3.** Given the exact cover

\[
\begin{align*}
T_{1,\mathbf{T}} &= \{x_1, p_{21}\}, T_{2,\mathbf{T}} = \{x_2, p_{12}, p_{41}\}, T_{3,\mathbf{F}} = \{x_3, p_{23}\}, \\
\{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}
\end{align*}
\]

we get the satisfying assigment $v(x_1) = \mathbf{T}, v(x_2) = \mathbf{T}, v(x_3) = \mathbf{F}$.
If we now consider the proposition is CNF given by

$$F_2 = \{C_1 = (x_1 \lor \overline{x_2}),\ C_2 = (\overline{x_1} \lor x_2 \lor x_3),\ C_3 = (x_2),\ C_4 = (\overline{x_2} \lor \overline{x_3} \lor x_4)\}$$

where we have added the boolean variable $x_4$ to clause $C_4$, then $U$ also contains $x_4$ and $p_{43}$ so we need to add the following subsets to $F$:

$$T_{4,F} = \{x_4,p_{43}\},\ T_{4,T} = \{x_4\},\ \{C_4,p_{43}\},\ \{p_{43}\}.$$  

The truth assignment $v(x_1) = \text{T}, v(x_2) = \text{T}, v(x_3) = \text{F}, v(x_4) = \text{T}$ satisfies $F_2$, so an exact cover $C$ is

$$T_{1,T} = \{x_1,p_{21}\},\ T_{2,T} = \{x_2,p_{12},p_{41}\},\ T_{3,F} = \{x_3,p_{23}\},\ T_{4,T} = \{x_4\},\ \{C_1,p_{11}\},\ \{C_2,p_{22}\},\ \{C_3,p_{31}\},\ \{C_4,p_{42}\},\ \{p_{43}\}.$$  

Observe that this time, because the truth assignment $v$ makes both literals corresponding to $p_{42}$ and $p_{43}$ true and since we picked $p_{42}$ to form the subset $\{C_4,p_{42}\}$, we need to add the singleton $\{p_{43}\}$ to $C$ to cover all elements of $U$.

(2) **Hamiltonian Cycle (for Directed Graphs)**

To prove that **Hamiltonian Cycle (for Directed Graphs)** is $\mathcal{NP}$-complete, we will reduce **Exact Cover** to it:

**Exact Cover $\leq_P$ Hamiltonian Cycle (for Directed Graphs)**

We need to find an algorithm working in polynomial time that converts an instance $(U,F)$ of **Exact Cover** to a directed graph $G = \tau(U,F)$ such that $G$ has a Hamiltonian cycle iff $(U,F)$ has an exact cover.

The construction of the graph $G$ uses a trick involving a small subgraph $\text{Gad}$ with 7 (distinct) nodes known as a *gadget* shown in Figure 10.7.

![Figure 10.7: A gadget Gad](image)

The crucial property of the graph $\text{Gad}$ is that if $\text{Gad}$ is a subgraph of a bigger graph $G$ in such a way that no edge of $G$ is incident to any of the nodes $u,v,w$ unless it
is one of the eight edges of \( G_{ad} \) incident to the nodes \( u, v, w \), then for any Hamiltonian cycle in \( G \), either the path \((a, u), (u, v), (v, w), (w, b)\) is traversed or the path \((c, w), (w, v), (v, u), (u, d)\) is traversed, but not both.

The reader should convince herself/himself that indeed, any Hamiltonian cycle that does not traverse either the subpath \((a, u), (u, v), (v, w), (w, b)\) from \( a \) to \( b \) or the subpath \((c, w), (w, v), (v, u), (u, d)\) from \( c \) to \( d \) will not traverse one of the nodes \( u, v, w \). Also, the fact that node \( v \) is traversed exactly once forces only one of the two paths to be traversed but not both. The reader should also convince herself/himself that a smaller graph does not guarantee the desired property.

It is convenient to use the simplified notation with a special type of edge labeled with the exclusive or sign \( \oplus \) between the “edges” between \( a \) and \( b \) and between \( d \) and \( c \), as shown in Figure 10.8.

![Figure 10.8: A shorthand notation for a gadget](image)

Whenever such a figure occurs, the actual graph is obtained by substituting a copy of the graph \( G_{ad} \) (the four nodes \( a, b, c, d \) must be distinct). This abbreviating device can be extended to the situation where we build gadgets between a given pair \((a, b)\) and several other pairs \((c_1, d_1), \ldots, (c_m, d_m)\), all nodes being distinct, as illustrated in Figure 10.9.

Either all three edges \((c_1, d_1), (c_2, d_2), (c_3, d_3)\) are traversed or the edge \((a, b)\) is traversed, and these possibilities are mutually exclusive.

The graph \( G = \tau(U, F) \) where \( U = \{u_1, \ldots, u_n\} \) (with \( n \geq 1 \)) and \( F = \{S_1, \ldots, S_m\} \) (with \( m \geq 1 \)) is constructed as follows:

The graph \( G \) has \( m + n + 2 \) nodes \( \{u_0, u_1, \ldots, u_n, S_0, S_1, \ldots, S_m\} \). Note that we have added two extra nodes \( u_0 \) and \( S_0 \). For \( i = 1, \ldots, m \), there are two edges \((S_{i-1}, S_i)_1\) and \((S_{i-1}, S_i)_2\) from \( S_{i-1} \) to \( S_i \). For \( j = 1, \ldots, n \), from \( u_{j-1} \) to \( u_j \), there are as many edges as there are sets \( S_i \in F \) containing the element \( u_j \). We can think of each edge between \( u_{j-1} \) and \( u_j \) as an occurrence of \( u_j \) in a uniquely determined set \( S_i \in F \); we denote this edge by \((u_{j-1}, u_j)_i\). We also have an edge from \( u_n \) to \( S_0 \) and an edge from \( S_m \) to \( u_0 \), thus “closing the cycle.”
What we have constructed so far is not a legal graph since it may have many parallel edges, but are going to turn it into a legal graph by pairing edges between the $u_j$’s and edges between the $S_i$’s. Indeed, since each edge $(u_{j-1}, u_j)_i$ between $u_{j-1}$ and $u_j$ corresponds to an occurrence of $u_j$ in some uniquely determined set $S_i \in \mathcal{F}$ (that is, $u_j \in S_i$), we put an exclusive-or edge between the edge $(u_{j-1}, u_j)_i$ and the edge $(S_{i-1}, S_i)_2$ between $S_{i-1}$ and $S_i$, which we call the long edge. The other edge $(S_{i-1}, S_i)_1$ between $S_{i-1}$ and $S_i$ (not paired with any other edge) is called the short edge. Effectively, we put a copy of the gadget graph $Gad$ with $a = u_{j-1}, b = u_j, c = S_{i-1}, d = S_i$ for any pair $(u_j, S_i)$ such that $u_j \in S_i$. The resulting object is indeed a directed graph with no parallel edges.

**Example 10.4.** The above construction is illustrated in Figure 10.10 for the instance of the exact cover problem given by

$$U = \{u_1, u_2, u_3, u_4\}, \quad \mathcal{F} = \{S_1 = \{u_3, u_4\}, \quad S_2 = \{u_2, u_3, u_4\}, \quad S_3 = \{u_1, u_2\}\}.$$ 

It remains to prove that $(U, \mathcal{F})$ has an exact cover iff the graph $G = \tau(U, \mathcal{F})$ has a Hamiltonian cycle. First, assume that $G$ has a Hamiltonian cycle. If so, for every $j$ some unique “edge” $(u_{j-1}, u_j)_i$ is traversed once (since every $u_j$ is traversed once), and by the exclusive-or nature of the gadget graphs, the corresponding long edge $(S_{i-1}, S_i)_2$ can’t be traversed, which means that the short edge $(S_{i-1}, S_i)_1$ is traversed. Consequently, if $\mathcal{C}$ consists of those subsets $S_i$ such that the short edge $(S_{i-1}, S_i)_1$ is traversed, then $\mathcal{C}$ consists of pairwise disjoint subsets whose union is $U$, namely $\mathcal{C}$ is an exact cover.

In our example, there is a Hamiltonian where the blue edges are traversed between the
Figure 10.10: The directed graph constructed from the data \((U, F)\) of Example 10.4

\(S_i\) nodes, and the red edges are traversed between the \(u_j\) nodes, namely

\[
\text{short } (S_0, S_1), \text{ long } (S_1, S_2), \text{ short } (S_2, S_3), \ (S_3, u_0), \\
(u_0, u_1)_3, \ (u_1, u_2)_3, \ (u_2, u_3)_1, \ (u_3, u_4)_1, \ (u_4, S_0).
\]

The subsets corresponding to the short \((S_{i-1}, S_i)\) edges are \(S_1\) and \(S_3\), and indeed \(C = \{S_1, S_3\}\) is an exact cover.

Note that the exclusive-or property of the gadgets implies the following: since the edge \((u_0, u_1)_3\) must be chosen to obtain a Hamiltonian, the long edge \((S_2, S_3)\) can’t be chosen, so the edge \((u_1, u_2)_3\) must be chosen, but then the edge \((u_1, u_2)_2\) is not chosen so the long edge \((S_1, S_2)\) must be chosen, so the edges \((u_2, u_3)_2\) and \((u_3, u_4)_2\) can’t be chosen, and thus edges \((u_2, u_3)_1\) and \((u_3, u_4)_1\) must be chosen.

Conversely, if \(C\) is an exact cover for \((U, F)\), then consider the path in \(G\) obtained by traversing each short edge \((S_{i-1}, S_i)_1\) for which \(S_i \in C\), each edge \((u_{j-1}, u_j)_i\) such that \(u_j \in S_i\), which means that this edge is connected by a \(\oplus\)-sign to the long edge \((S_{i-1}, S_i)_2\) (by construction, for each \(u_j\) there is a unique such \(S_i\)), and the edges \((u_n, S_0)\) and \((S_m, u_0)\), then we obtain a Hamiltonian cycle.
In our example, the exact cover \( \mathcal{C} = \{S_1, S_3\} \) yields the Hamiltonian
\[
\text{short } (S_0, S_1), \text{ long } (S_1, S_2), \text{ short } (S_2, S_3), \text{ (}S_3, u_0\text{)}, \\
(u_0, u_1)_3, (u_1, u_2)_3, (u_2, u_3)_1, (u_3, u_4)_1, (u_4, S_0)
\]
that we encountered earlier.

(3) **Hamiltonian Cycle (for Undirected Graphs)**

To show that **Hamiltonian Cycle (for Undirected Graphs)** is \( \mathcal{NP} \)-complete we reduce **Hamiltonian Cycle (for Directed Graphs)** to it:

**Hamiltonian Cycle (for Directed Graphs) \( \leq_P \) Hamiltonian Cycle (for Undirected Graphs)**

Given any directed graph \( G = (V, E) \) we need to construct in polynomial time an undirected graph \( \tau(G) = G' = (V', E') \) such that \( G \) has a (directed) Hamiltonian cycle iff \( G' \) has a (undirected) Hamiltonian cycle. This is easy. We make three distinct copies \( v_0, v_1, v_2 \) of every node \( v \in V \) which we put in \( V' \), and for every edge \((u, v) \in E\) we create five edges \{\(u_0, u_1\}, \{u_1, u_2\}, \{u_2, v_0\}, \{v_0, v_1\}, \{v_1, v_2\}\} which we put in \( E' \), as illustrated in the diagram shown in Figure 10.11.

![Figure 10.11: Conversion of a directed graph into an undirected graph](image)

The crucial point about the graph \( G' \) is that although there may be several edges adjacent to a node \( u_0 \) or a node \( u_2 \), the only way to reach \( u_1 \) from \( u_0 \) is through the edge \{\(u_0, u_1\}\} and the only way to reach \( u_1 \) from \( u_2 \) is through the edge \{\(u_1, u_2\}\}.

Suppose there is a Hamiltonian cycle in \( G' \). If this cycle arrives at a node \( u_0 \) from the node \( u_1 \), then by the above remark, the previous node in the cycle must be \( u_2 \). Then, the predecessor of \( u_2 \) in the cycle must be a node \( v_0 \) such that there is an edge \{\(u_2, v_0\}\} in \( G' \) arising from an edge \((u, v)\) in \( G \). The nodes in the cycle in \( G' \) are traversed in the order \((v_0, u_2, u_1, u_0)\) where \( v_0 \) and \( u_2 \) are traversed in the opposite order in which they occur as the endpoints of the edge \((u, v)\) in \( G \). If so, consider the reverse of our Hamiltonian cycle in \( G' \), which is also a Hamiltonian cycle since \( G' \) is unoriented. In this cycle, we go from \( u_0 \) to \( u_1 \), then to \( u_2 \), and finally to \( v_0 \). In \( G \), we traverse the edge from \( u \) to \( v \). In order for the cycle in \( G' \) to be Hamiltonian, we must continue by visiting \( v_1 \) and \( v_2 \) since otherwise \( v_1 \) is never traversed. Now, the next node \( w_0 \) in the Hamiltonian cycle in \( G' \) corresponds to an edge \((v, w)\) in \( G \), and by repeating our reasoning we see that our Hamiltonian cycle in \( G' \) determines a Hamiltonian cycle in \( G \). We leave it as an easy exercise to check that a Hamiltonian cycle in \( G \) yields a Hamiltonian cycle in \( G' \).
10.2. PROOFS OF \( \mathcal{NP} \)-COMPLETENESS

(4) Traveling Salesman Problem

To show that the Traveling Salesman Problem is \( \mathcal{NP} \)-complete, we reduce the Hamiltonian Cycle Problem (Undirected Graphs) to it:

Hamiltonian Cycle Problem (Undirected Graphs) \( \leq_p \) Traveling Salesman Problem

This is a fairly easy reduction.

Given an undirected graph \( G = (V,E) \), we construct an instance \( \tau(G) = (D,B) \) of the traveling salesman problem so that \( G \) has a Hamiltonian cycle iff the traveling salesman problem has a solution. If we let \( n = |V| \), we have \( n \) cities and the matrix \( D = (d_{ij}) \) is defined as follows:

\[
 d_{ij} = \begin{cases} 
 0 & \text{if } i = j \\
 1 & \text{if } \{v_i,v_j\} \in E \\
 2 & \text{otherwise.} 
\end{cases}
\]

We also set the budget \( B \) as \( B = n \).

Any tour of the cities has cost equal to \( n \) plus the number of pairs \( (v_i,v_j) \) such that \( i \neq j \) and \( \{v_i,v_j\} \) is not an edge of \( G \). It follows that a tour of cost \( n \) exists iff there are no pairs \( (v_i,v_j) \) of the second kind iff the tour is a Hamiltonian cycle.

The reduction from Hamiltonian Cycle Problem (Undirected Graphs) to the Traveling Salesman Problem is quite simple, but a direct reduction of say Satisfiability to the Traveling Salesman Problem is hard. By breaking this reduction into several steps made it simpler to achieve.

(5) Independent Set

To show that Independent Set is \( \mathcal{NP} \)-complete, we reduce Exact 3-Satisfiability to it:

Exact 3-Satisfiability \( \leq_p \) Independent Set

Recall that in Exact 3-Satisfiability every clause \( C_i \) has exactly three literals \( L_{i1}, L_{i2}, L_{i3} \).

Given a set \( F = \{C_1, \ldots, C_m\} \) of \( m \geq 2 \) such clauses, we construct in polynomial time an undirected graph \( G = (V,E) \) such that \( F \) is satisfiable iff \( G \) has an independent set \( C \) with at least \( K = m \) nodes.

For every \( i \) (1 \( \leq i \leq m \)), we have three nodes \( c_{i1}, c_{i2}, c_{i3} \) corresponding to the three literals \( L_{i1}, L_{i2}, L_{i3} \) in clause \( C_i \), so there are \( 3m \) nodes in \( V \). The “core” of \( G \) consists of \( m \) triangles, one for each set \( \{c_{i1}, c_{i2}, c_{i3}\} \). We also have an edge \( \{c_{ik}, c_{j\ell}\} \) iff \( L_{ik} \) and \( L_{j\ell} \) are complementary literals.
Example 10.5. Let $F$ be the set of clauses

$$F = \{C_1 = (x_1 \lor \overline{x_2} \lor x_3), C_2 = (\overline{x_1} \lor \overline{x_2} \lor x_3), C_3 = (\overline{x_1} \lor \overline{x_2} \lor \overline{x_3}), C_4 = (x_1 \lor x_2 \lor x_3)\}.$$ 

The graph $G$ associated with $F$ is shown in Figure 10.12.

![Figure 10.12: The graph constructed from the clauses of Example 10.5](image)

It remains to show that the construction works. Since any three nodes in a triangle are connected, an independent set $C$ can have at most one node per triangle and thus has at most $m$ nodes. Since the budget is $K = m$, we may assume that there is an independent set with $m$ nodes. Define a (partial) truth assignment by

$$v(x_i) = \begin{cases} 
T & \text{if } L_{jk} = x_i \text{ and } c_{jk} \in C \\
F & \text{if } L_{jk} = \overline{x_i} \text{ and } c_{jk} \in C.
\end{cases}$$

Since the non-triangle edges in $G$ link nodes corresponding to complementary literals and nodes in $C$ are not connected, our truth assignment does not assign clashing truth values to the variables $x_i$. Not all variables may receive a truth value, in which case we assign an arbitrary truth value to the unassigned variables. This yields a satisfying assignment for $F$.

In Example 10.5, the set $C = \{c_{11}, c_{22}, c_{32}, c_{41}\}$ corresponding to the nodes shown in red in Figure 10.12 form an independent set, and they induce the partial truth assignment $v(x_1) = T, v(x_2) = F$. The variable $x_3$ can be assigned an arbitrary value, say $v(x_3) = F$, and $v$ is indeed a satisfying truth assignment for $F$.

Conversely, if $v$ is a truth assignment for $F$, then we obtain an independent set $C$ of size $m$ by picking for each clause $C_i$ a node $c_{ik}$ corresponding to a literal $L_{ik}$ whose value under $v$ is $T$. 
(6) **Clique**

To show that **Clique** is \(NP\)-complete, we reduce **Independent Set** to it:

**Independent Set \(\leq_P\) Clique**

The key the reduction is the notion of the complement of an undirected graph \(G = (V, E)\). The complement \(G^c = (V, E^c)\) of the graph \(G = (V, E)\) is the graph with the same set of nodes \(V\) as \(G\) but there is an edge \(\{u, v\}\) (with \(u \neq v\)) in \(E^c\) iff \(\{u, v\} \notin E\). Then, it is not hard to check that there is a bijection between maximum independent sets in \(G\) and maximum cliques in \(G^c\). The reduction consists in constructing from a graph \(G\) its complement \(G^c\), and then \(G\) has an independent set iff \(G^c\) has a clique.

This construction is illustrated in Figure 10.13, where a maximum independent set in the graph \(G\) is shown in blue and a maximum clique in the graph \(G^c\) is shown in red.

![Figure 10.13: A graph (left) and its complement (right)](image)

(7) **Node Cover**

To show that **Node Cover** is \(NP\)-complete, we reduce **Independent Set** to it:

**Independent Set \(\leq_P\) Node Cover**

This time the crucial observation is that if \(N\) is an independent set in \(G\), then the complement \(C = V − N\) of \(N\) in \(V\) is a node cover in \(G\). Thus there is an independent set of size at least \(K\) iff there is a node cover of size at most \(n − K\) where \(n = |V|\) is the number of nodes in \(V\). The reduction leaves the graph unchanged and replaces \(K\) by \(n − K\). An example is shown in Figure 10.14 where an independent set is shown in blue and a node cover is shown in red.

(8) **Knapsack** (also called **Subset sum**)

To show that **Knapsack** is \(NP\)-complete, we reduce **Exact Cover** to it:

**Exact Cover \(\leq_P\) Knapsack**
Figure 10.14: An independent set (left) and a node cover (right)

Given an instance \((U, \mathcal{F})\) of set cover with \(U = \{u_1, \ldots, u_n\}\) and \(\mathcal{F} = \{S_1, \ldots, S_m\}\), a family of subsets of \(U\), we need to produce in polynomial time an instance \(\tau(U, \mathcal{F})\) of the knapsack problem consisting of \(k\) nonnegative integers \(a_1, \ldots, a_k\) and another integer \(K > 0\) such that there is a subset \(I \subseteq \{1, \ldots, k\}\) such that \(\sum_{i \in I} a_i = K\) iff there is an exact cover of \(U\) using subsets in \(\mathcal{F}\).

The trick here is the relationship between set union and integer addition.

**Example 10.6.** Consider the exact cover problem given by \(U = \{u_1, u_2, u_3, u_4\}\) and
\[
\mathcal{F} = \{S_1 = \{u_3, u_4\}, S_2 = \{u_2, u_3, u_4\}, S_3 = \{u_1, u_2\}\}.
\]

We can represent each subset \(S_j\) by a binary string \(a_j\) of length 4, where the \(i\)th bit from the left is 1 iff \(u_i \in S_j\), and 0 otherwise. In our example
\[
\begin{align*}
a_1 &= 0011 \\
a_2 &= 0111 \\
a_3 &= 1100.
\end{align*}
\]

Then, the trick is that some family \(\mathcal{C}\) of subsets \(S_j\) is an exact cover if the sum of the corresponding numbers \(a_j\) adds up to \(1111 = 2^4 - 1 = K\). For example,
\[
\mathcal{C} = \{S_1 = \{u_3, u_4\}, S_3 = \{u_1, u_2\}\}
\]
is an exact cover and
\[
a_1 + a_3 = 0011 + 1100 = 1111.
\]

Unfortunately, there is a problem with this encoding which has to do with the fact that addition may involve carry. For example, assuming four subsets and the universe \(U = \{u_1, \ldots, u_6\}\),
\[
11 + 13 + 15 + 24 = 63,
\]
in binary
\[
001011 + 001101 + 001111 + 011000 = 111111,
\]
but if we convert these binary strings to the corresponding subsets we get the subsets

\[ S_1 = \{u_3, u_5, u_6\} \]
\[ S_2 = \{u_3, u_4, u_6\} \]
\[ S_3 = \{u_3, u_4, u_5, u_6\} \]
\[ S_4 = \{u_2, u_3\} \]

which are not disjoint and do not cover \( U \).

The fix is surprisingly simple: use base \( m \) (where \( m \) is the number of subsets in \( F \)) instead of base 2.

**Example 10.7.** Consider the exact cover problem given by \( U = \{u_1, u_2, u_3, u_4, u_5, u_6\} \) and \( F \) given by

\[ S_1 = \{u_3, u_5, u_6\} \]
\[ S_2 = \{u_3, u_4, u_6\} \]
\[ S_3 = \{u_3, u_4, u_5, u_6\} \]
\[ S_4 = \{u_2, u_3\} \]
\[ S_5 = \{u_1, u_2, u_4\} \]

In base \( m = 5 \), the numbers corresponding to \( S_1, \ldots, S_5 \) are

\[ a_1 = 001011 \]
\[ a_2 = 001101 \]
\[ a_3 = 001111 \]
\[ a_4 = 011000 \]
\[ a_5 = 110100 \]

This time,

\[ a_1 + a_2 + a_3 + a_4 = 001011 + 001101 + 001111 + 011000 = 014223 \neq 111111 \]

so \( \{S_1, S_2, S_3, S_4\} \) is not a solution. However

\[ a_1 + a_5 = 001011 + 110100 = 111111 \]

and \( C = \{S_1, S_5\} \) is an exact cover.

Thus, given an instance \((U, F)\) of **Exact Cover** where \( U = \{u_1, \ldots, u_n\} \) and \( F = \{S_1, \ldots, S_m\} \) the reduction to **Knapsack** consists in forming the \( m \) numbers \( a_1, \ldots, a_m \) (each of \( n \) bits) encoding the subsets \( S_j \), namely \( a_{ji} = 1 \) iff \( u_i \in S_j \), else 0, and to let
$K = 1 + m^2 + \cdots + m^{n-1}$, which is represented in base $m$ by the string $11\cdots11$. In testing whether $\sum_{i \in I} a_i = K$ for some subset $I \subseteq \{1, \ldots, m\}$, we use arithmetic in base $m$.

If a candidate solution $C$ involves at most $m - 1$ subsets, then since the corresponding numbers are added in base $m$, a carry can never happen. If the candidate solution involves all $m$ subsets, then $a_1 + \cdots + a_m = K$ iff $\mathcal{F}$ is a partition of $U$, since otherwise some bit in the result of adding up these $m$ numbers in base $m$ is not equal to 1, even if a carry occurs.

(9) Inequivalence of $\ast$-free Regular Expressions

To show that Inequivalence of $\ast$-free Regular Expressions is $\mathcal{NP}$-complete, we reduce the Satisfiability Problem to it:

Satisfiability Problem $\leq_p$ Inequivalence of $\ast$-free Regular Expressions

We already argued that Inequivalence of $\ast$-free Regular Expressions is in $\mathcal{NP}$ because if $R$ is a $\ast$-free regular expression, then for every string $w \in \mathcal{L}[R]$ we have $|w| \leq |R|$. The above observation shows that if $R_1$ and $R_2$ are $\ast$-free and if there is a string $w \in (\mathcal{L}[R_1] - \mathcal{L}[R_2]) \cup (\mathcal{L}[R_2] - \mathcal{L}[R_1])$, then $|w| \leq |R_1| + |R_2|$, so we can indeed check this in polynomial time. It follows that the inequivalence problem for $\ast$-free regular expressions is in $\mathcal{NP}$.

We reduce the Satisfiability Problem to the Inequivalence of $\ast$-free Regular Expressions as follows. For any set of clauses $P = C_1 \land \cdots \land C_p$, if the propositional variables occurring in $P$ are $x_1, \ldots, x_n$, we produce two $\ast$-free regular expressions $R, S$ over $\Sigma = \{0, 1\}$, such that $P$ is satisfiable iff $L_R \neq L_S$. The expression $S$ is actually

$$S = (0 + 1)(0 + 1)^n \cdots (0 + 1).$$

The expression $R$ is of the form

$$R = R_1 + \cdots + R_p,$$

where $R_i$ is constructed from the clause $C_i$ in such a way that $L_{R_i}$ corresponds precisely to the set of truth assignments that falsify $C_i$; see below.

Given any clause $C_i$, let $R_i$ be the $\ast$-free regular expression defined such that, if $x_j$ and $\overline{x}_j$ both belong to $C_i$ (for some $j$), then $R_i = \emptyset$, else

$$R_i = R_i^1 \cdot R_i^2 \cdots R_i^n,$$
where $R^j_i$ is defined by

$$R^j_i = \begin{cases} 
0 & \text{if } x_j \text{ is a literal of } C_i \\
1 & \text{if } \overline{x_j} \text{ is a literal of } C_i \\
(0 + 1) & \text{if } x_j \text{ does not occur in } C_i.
\end{cases}$$

Clearly, all truth assignments that falsify $C_i$ must assign $\mathbf{F}$ to $x_j$ if $x_j \in C_i$ or assign $\mathbf{T}$ to $x_j$ if $\overline{x_j} \in C_i$. Therefore, $L_{R_i}$ corresponds to the set of truth assignments that falsify $C_i$ (where 1 stands for $\mathbf{T}$ and 0 stands for $\mathbf{F}$) and thus, if we let

$$R = R_1 + \cdots + R_p,$$

then $L_R$ corresponds to the set of truth assignments that falsify $P = C_1 \land \cdots \land C_p$. Since $L_S = \{0, 1\}^n$ (all binary strings of length $n$), we conclude that $L_R \neq L_S$ iff $P$ is satisfiable. Therefore, we have reduced the Satisfiability Problem to our problem and the reduction clearly runs in polynomial time. This proves that the problem of deciding whether $L_R \neq L_S$, for any two $*$-free regular expressions $R$ and $S$ is $\mathcal{NP}$-complete.

(10) **0-1 integer programming problem**

It is easy to check that the problem is in $\mathcal{NP}$.

To prove that the is $\mathcal{NP}$-complete we reduce the bounded-tiling problem to it:

**bounded-tiling problem $\leq_P$ 0-1 integer programming problem**

Given a tiling problem, $((T, V, H), \hat{s}, \sigma_0)$, we create a 0-1-valued variable $x_{mnt}$, such that $x_{mnt} = 1$ iff tile $t$ occurs in position $(m, n)$ in some tiling. Write equations or inequalities expressing that a tiling exists and then use “slack variables” to convert inequalities to equations. For example, to express the fact that every position is tiled by a single tile, use the equation

$$\sum_{t \in T} x_{mnt} = 1,$$

for all $m, n$ with $1 \leq m \leq 2s$ and $1 \leq n \leq s$. We leave the rest as as exercise.

### 10.3 Succinct Certificates, $\text{coNP}$, and $\text{EXP}$

All the problems considered in Section 10.1 share a common feature, which is that for each problem, a solution is produced nondeterministically (an exact cover, a directed Hamiltonian cycle, a tour of cities, an independent set, a node cover, a clique etc.), and then this candidate solution is checked deterministically and in polynomial time. The candidate solution is a string called a *certificate* (or *witness*).

It turns out that membership on $\mathcal{NP}$ can be defined in terms of certificates. To be a certificate, a string must satisfy two conditions:
1. It must be *polynomially succinct*, which means that its length is at most a polynomial in the length of the input.

2. It must be *checkable* in polynomial time.

All “yes” inputs to a problem in $\mathcal{NP}$ must have at least one certificate, while all “no” inputs must have none.

The notion of certificate can be formalized using the notion of a polynomially balanced language.

**Definition 10.3.** Let $\Sigma$ be an alphabet, and let “;” be a symbol not in $\Sigma$. A language $L' \subseteq \Sigma^*; \Sigma^*$ is said to be *polynomially balanced* if there exists a polynomial $p(X)$ such that for all $x, y \in \Sigma^*$, if $x; y \in L'$ then $|y| \leq p(|x|)$.

Suppose $L'$ is a polynomially balanced language and that $L' \in \mathcal{P}$. Then we can consider the language

$$L = \{ x \in \Sigma^* | (\exists y \in \Sigma^*)(x; y \in L') \}.$$  

The intuition is that for each $x \in L$, the set

$$\{ y \in \Sigma^* | x; y \in L' \}$$

is the set of certificates of $x$. For every $x \in L$, a Turing machine can nondeterministically guess one of its certificates $y$, and then use the deterministic Turing machine for $L'$ to check in polynomial time that $x; y \in L'$. Note that, by definition, strings not in $L$ have no certificate. It follows that $L \in \mathcal{NP}$.

Conversely, if $L \in \mathcal{NP}$ and the alphabet $\Sigma$ has at least two symbols, we can encode the paths in the computation tree for every input $x \in L$, and we obtain a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ in $\mathcal{P}$ such that

$$L = \{ x \in \Sigma^* | (\exists y \in \Sigma^*)(x; y \in L') \}.$$  

The details of this construction are left as an exercise. In summary, we obtain the following theorem.

**Theorem 10.1.** Let $L \subseteq \Sigma^*$ be a language over an alphabet $\Sigma$ with at least two symbols, and let “;” be a symbol not in $\Sigma$. Then $L \in \mathcal{NP}$ iff there is a polynomially balanced language $L' \subseteq \Sigma^*; \Sigma^*$ such that $L' \in \mathcal{P}$ and

$$L = \{ x \in \Sigma^* | (\exists y \in \Sigma^*)(x; y \in L') \}.$$  

A striking illustration of the notion of succinct certificate is illustrated by the set of *composite* integers, namely those natural numbers $n \in \mathbb{N}$ that can be written as the product $pq$ of two numbers $p, q \geq 2$ with $p, q \in \mathbb{N}$. For example, the number

$$4, 294, 967, 297$$
is a composite!

This is far from obvious, but if an oracle gives us the certificate \{6, 700, 417, 641\}, it is easy to carry out in polynomial time the multiplication of these two numbers and check that it is equal to 4,294,967,297. Finding a certificate is usually (very) hard, but checking that it works is easy. This is the point of certificates.

We conclude this section with a brief discussion of the complexity classes coNP and EXP.

By definition, 

$$coNP = \{ L \mid L \in NP \},$$

that is, coNP consists of all complements of languages in NP. Since \( P \subseteq NP \) and \( P \) is closed under complementation,

$$P \subseteq coNP,$$

but nobody knows whether \( NP \) is closed under complementation, that is, nobody knows whether \( NP = coNP \).

What can be shown is that if \( NP \neq coNP \) then \( P \neq NP \).

A natural instance of a problem in coNP is the unsatisfiability problem for propositions, namely deciding that a proposition \( P \) has no satisfying assignment. Since a proposition \( P \) is valid iff \( \neg P \) is unsatisfiable, the validity problem for propositions is in coNP. Despite the fact that this problem has been extensively studied, not much is known about its exact complexity.

The class EXP is defined as follows.

**Definition 10.4.** A deterministic Turing machine \( M \) is said to be exponentially bounded if there is a polynomial \( p(X) \) such that for every input \( x \in \Sigma^* \), there is no ID \( ID_n \) such that

$$ID_0 \vdash ID_1 \vdash \ldots \vdash ID_n, \quad \text{with} \quad n > 2^{p(|x|)}.$$

The class EXP is the class of all languages that are accepted by some exponentially bounded deterministic Turing machine.

**Remark:** We can also define the class NEXP as in Definition 10.4, except that we allow nondeterministic Turing machines.

One of the interesting features of EXP is that it contains NP.

**Theorem 10.2.** We have the inclusion \( NP \subseteq EXP \).

**Sketch of proof.** Let \( M \) be some nondeterministic Turing machine accepting \( L \) in polynomial time bounded by \( p(X) \). We can construct a deterministic Turing machine \( M' \) that operates as follows: for every input \( x \), \( M' \) simulates \( M \) on all computations of length 1, then on all possible computations of length 2, and so on, up to all possible computations of length
At this point, either an accepting computation has been discovered or all computations have halted rejecting. We claim that $M'$ operates in time bounded by $2^q(|x|)$ for some polynomial $q(X)$. First, let $r$ be the degree of nondeterminism of $M$, that is, the maximum number of triples $(b, m, q)$ such that a quintuple $(p, q, b, m, q)$ is an instruction of $M$. Then to simulate a computation of $M$ of length $\ell$, $M'$ needs $O(\ell)$ steps—to copy the input, to produce a string $c$ in $\{1, \ldots, r\}^\ell$, and so simulate $M$ according to the choices specified by $c$. It follows that $M'$ can carry out the simulation of $M$ on an input $x$ in

$$
\sum_{\ell=1}^{p(|x|)+1} r^\ell \leq (r + 1)^{p(|x|)+1}
$$

steps. Including the $O(\ell)$ extra steps for each $\ell$, we obtain the bound $(r + 2)^{p(|x|)+1}$. Then, we can pick a constant $k$ such that $2^k > r + 2$, and with $q(X) = k(p(X) + 1)$, we see that $M'$ operates in time bounded by $2^q(|x|)$.

It is also immediate to see that $\mathcal{EXP}$ is closed under complementation. Furthermore the strict inclusion $\mathcal{P} \subset \mathcal{EXP}$ holds.

**Theorem 10.3.** We have the strict inclusion $\mathcal{P} \subset \mathcal{EXP}$.

*Sketch of proof.* We use a diagonalization argument to produce a language $E$ such that $E \notin \mathcal{P}$, yet $E \in \mathcal{EXP}$. We need to code a Turing machine as a string, but this can certainly be done using the techniques of Chapter 6. Let $\#(M)$ be the code of Turing machine $M$. Define $E$ as

$$
E = \{\#(M)x \mid M \text{ accepts input } x \text{ after at most } 2^{|x|} \text{ steps}\}.
$$

We claim that $E \notin \mathcal{P}$. We proceed by contradiction. If $E \in \mathcal{P}$, then so is the language $E_1$ given by

$$
E_1 = \{\#(M) \mid M \text{ accepts } \#(M) \text{ after at most } 2^{\#(M)} \text{ steps}\}.
$$

Since $\mathcal{P}$ is closed under complementation, we also have $\overline{E_1} \in \mathcal{P}$. Let $M^*$ be a deterministic Turing machine accepting $\overline{E_1}$ in time $p(X)$, for some polynomial $p(X)$. Since $p(X)$ is a polynomial, there is some $n_0$ such that $p(n) \leq 2^n$ for all $n \geq n_0$. We may also assume that $|\#(M^*)| \geq n_0$, since if not we can add $n_0$ "dead states" to $M^*$.

Now, what happens if we run $M^*$ on its own code $\#(M^*)$?

It is easy to see that we get a contradiction, namely $M^*$ accepts $\#(M^*)$ iff $M^*$ rejects $\#(M^*)$. We leave this verification as an exercise.

In conclusion, $\overline{E_1} \notin \mathcal{P}$, which in turn implies that $E \notin \mathcal{P}$.

It remains to prove that $E \in \mathcal{EXP}$. This is because we can construct a Turing machine that can in exponential time simulate any Turing machine $M$ on input $x$ for $2^{|x|}$ steps. 

\[\square\]
In summary, we have the chain of inclusions

\[ \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{EXP}, \]

where the left inclusion and the right inclusion are both open problems, but we know that at least one of these two inclusions is strict.
Bibliography
