Quantifiers in TIME and SPACE: computational complexity of generalized quantifiers in natural language
Szymanik, J.K.

Link to publication

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.

UvA-DARE is a service provided by the library of the University of Amsterdam (http://dare.uva.nl)

Download date: 04 Oct 2018
Chapter 2
Mathematical Prerequisites

This chapter describes the notation and basic terminology from generalized quantifier theory and computation theory. We focus only on concepts which will be explored in the following parts of the thesis. We assume some familiarity with the basics of first-order and second-order logic (see e.g. Ebbinghaus et al., 1996).

2.1 Basic Notation

Let \( A, B \) be sets. We will write \( \emptyset \) for the *empty set*, \( \text{card}(A) \) to denote the *cardinality of the set* \( A \) and \( \mathcal{P}(B) \) for the *power set* of \( B \). The operations \( \cup, \cap, \setminus \), and the relation \( \subseteq \) on sets are defined as usual. The set \( A \times B = \{ (a, b) \mid a \in A \text{ and } b \in B \} \) denotes the *Cartesian product* of \( A \) and \( B \), where \( (a, b) \) is the ordered pair. If \( R \) is a relation then by \( R^{-1} \) we denote its inverse. *The set of natural numbers* is denoted by \( \omega \). If \( k \in \omega \) then \( A^k \) denotes \( \underbrace{A \times \ldots \times A}_k \).

We will write \( \mathbb{Q} \) for *the set of rational numbers*. If \( q \in \mathbb{Q} \) we write \([q]\) for the ceiling function of \( q \), i.e. the smallest integer greater than \( q \).

Let \( \varphi \) and \( \psi \) be formulae. We write \( \neg \varphi, \varphi \land \psi, \varphi \lor \psi, \varphi \Rightarrow \psi, \varphi \iff \psi, \exists x \varphi(x), \text{ and } \forall x \varphi(x) \) with the usual meaning. We will denote first-order logic (elementary logic) by FO and second-order logic by SO.

Now, let us recall the definition of the *hierarchy of second-order formulae*.

2.1.1. Definition. The class \( \Sigma^1_n \) is identical to the class \( \Pi^1_n \) and both contain (all and only) the first-order formulae. The class \( \Sigma^1_{n+1} \) is the set of formulæ of the following form: \( \exists P_1 \ldots \exists P_k \psi \), where \( \psi \in \Pi^1_n \). The class \( \Pi^1_{n+1} \) consists of formulæ of the form: \( \forall P_1 \ldots \forall P_k \psi \), where \( \psi \in \Sigma^1_n \). We additionally assume that all formulæ equivalent to some \( \Sigma^1_n \) (or \( \Pi^1_n \)) formula are also in \( \Sigma^1_n \) (respectively \( \Pi^1_n \)).
In the thesis a *vocabulary* is a finite set consisting of relation symbols (predicates) with assigned arities. Let $\tau = \{R_1, \ldots, R_k\}$ be a vocabulary, where for each $i$, $n_i$ is the arity of $R_i$. Then a $\tau$-model will be a structure of the following form: $M = (M, R_1, \ldots, R_k)$, where $M$ is the universe of model $M$ and $R_i \subseteq M$ is an $n_i$-ary relation over $M$, for $1 \leq i \leq k$. If $\varphi$ is a $\tau$-sentence (a sentence over the vocabulary $\tau$) then the class of $\tau$-models of $\varphi$ is denoted by $\text{Mod}(\varphi)$. We will sometimes write $R_i^M$ for relations to differentiate them from the corresponding predicates $R_i$.

### 2.2 Generalized Quantifier Theory

Generalized quantifiers are one of the basic tools of today’s linguistics and their mathematical properties have been extensively studied since the fifties (see Peters and Westerståhl, 2006, for a recent overview). In its simplest form generalized quantifier theory assigns meanings to statements by defining the semantics of the quantifiers occurring in them. For instance, the quantifiers “every”, “some”, “at least 7”, “an even number of”, and “most” build the following sentences.

1. Every poet has low self-esteem.
2. Some dean danced nude on the table.
3. At least 7 grad students prepared presentations.
4. An even number of the students saw a ghost.
5. Most of the students think they are smart.
6. Less than half of the students received good marks.

What is the semantics assigned to these quantifiers? Formally they are treated as relations between subsets of the universe. For instance, in sentence (1) “every” is a binary relation between the set of poets and the set of people having low self-esteem. Following the natural linguistic intuition we will say that sentence (1) is true if and only if the set of poets is included in the set of people having low self-esteem. Hence, the quantifier “every” corresponds in this sense to the inclusion relation.

Let us now have a look at sentence (4). It is true if and only if the intersection of the set of all students with the set of people who saw a ghost is of even cardinality. That is, this quantifier says something about the parity of the intersection of two sets.

Finally, let us consider example (5). Let us assume that the quantifier “most” simply means “more than half”\(^1\). Hence, sentence (5) is true if and only if the

\(^1\)We would say that the stronger, but vague, meaning that seems more natural is a result of pragmatics.
cardinality of the set of students who think they are smart is greater than the cardinality of the set of students who do not think they are smart. That is, the quantifier “most” expresses that these two kinds of student exist in a specific proportion.

2.2.1 Two Equivalent Concepts of Generalized Quantifiers

Frege was one of the major figures in forming the modern concept of quantification. In his Begriffsschrift (1879) he made a distinction between bound and free variables and treated quantifiers as well-defined, denoting entities. He thought of quantifiers as third-order objects — relations between subsets of a given fixed universe. This way of thinking about quantifiers is still current, particularly in linguistics. However, historically speaking the notion of a generalized quantifier was formulated for the first time in a different, although equivalent, way: generalized quantifiers were defined as classes of models closed under isomorphisms. Firstly, in a seminal paper of Andrzej Mostowski (1957) the notions of existential and universal quantification were extended to the concept of a monadic generalized quantifier binding one variable in one formula, and later this was generalized to arbitrary types by Per Lindström (1966). Below we give the formal definition.

2.2.1. DEFINITION. Let \( t = (n_1, \ldots, n_k) \) be a \( k \)-tuple of positive integers. A Lindström quantifier of type \( t \) is a class \( Q \) of models of a vocabulary \( \tau_t = \{ R_1, \ldots, R_k \} \), such that \( R_i \) is \( n_i \)-ary for \( 1 \leq i \leq k \), and \( Q \) is closed under isomorphisms, i.e. if \( M \) and \( M' \) are isomorphic, then

\[
(M \in Q \iff M' \in Q).
\]

2.2.2. DEFINITION. If in the above definition for all \( i: n_i \leq 1 \), then we say that a quantifier is monadic, otherwise we call it polyadic.

2.2.3. EXAMPLE. Let us explain this definition further by giving a few examples. Sentence (1) is of the form Every \( A \) is \( B \), where \( A \) stands for poets and \( B \) for people having low self-esteem. As we explained above the sentence is true if and only if \( A \subseteq B \). Therefore, according to the definition, the quantifier “every” is of type \((1, 1)\) and corresponds to the class of models \((M, A, B)\) in which \( A \subseteq B \). For the same reasons the quantifier “an even number of” corresponds to the class of models in which the cardinality of \( A \cap B \) is an even number. Finally, let us consider the quantifier “most” of type \((1, 1)\). As we mentioned before the sentence Most As are \( B \) is true if and only if \( \text{card}(A \cap B) > \text{card}(A - B) \) and therefore the quantifier corresponds to the class of models where this inequality holds.

Therefore, formally speaking:
\( \forall = \{(M, A) \mid A = M\} \).
\( \exists = \{(M, A) \mid A \subseteq M \text{ and } A \neq \emptyset\} \).
\( \text{Every} = \{(M, A, B) \mid A, B \subseteq M \text{ and } A \subseteq B\} \).
\( \text{Even} = \{(M, A, B) \mid A, B \subseteq M \text{ and } \text{card}(A \cap B) \text{ is even}\} \).
\( \text{Most} = \{(M, A, B) \mid A, B \subseteq M \text{ and } \text{card}(A \cap B) > \text{card}(A - B)\} \).

The first two examples are the standard first-order universal and existential quantifiers, both of type (1). They are classes of models with one unary predicate such that the extension of the predicate is equal to the whole universe in case of the universal quantifier and is nonempty in case of the existential quantifier.

Why do we assume that these classes are closed under isomorphisms? Simply put, this guarantees that the quantifiers are topic neutral. The quantifier “most” means exactly the same when applied to people as when applied to natural numbers.

Let us now give the definition of a generalized quantifier. This definition is commonly used in linguistics as opposed to the previous one which finds its applications rather in logic.

2.2.4. Definition. A generalized quantifier \( Q \) of type \( t = (n_1, \ldots, n_k) \) is a functor assigning to every set \( M \) a \( k \)-ary relation \( Q_M \) between relations on \( M \) such that if \( (R_1, \ldots, R_k) \in Q_M \) then \( R_i \) is an \( n_i \)-ary relation on \( M \), for \( i = 1, \ldots, k \). Additionally, \( Q \) is preserved by bijections, i. e., if \( f : M \rightarrow M' \) is a bijection then \( (R_1, \ldots, R_k) \in Q_M \) if and only if \( (fR_1, \ldots, fR_k) \in Q_{M'} \), for every relation \( R_1, \ldots, R_k \) on \( M \), where \( fR = \{(f(x_1), \ldots, f(x_i)) \mid (x_1, \ldots, x_i) \in R\} \), for \( R \subseteq M \).

In other words, a generalized quantifier \( Q \) is a functional relation associating with each model \( M \) a relation between relations on its universe, \( M \). Hence, if we fix a model \( M \), then we can treat a generalized quantifier as a relation between relations over the universe, and this is the familiar notion from natural language semantics.

Notice that we have the following equivalence between a Lindström quantifier and a generalized quantifier:

\((M, R_1, \ldots, R_k) \in Q \iff Q_M(R_1, \ldots, R_k), \text{where } R_i \subseteq M^{n_i}\).

For instance, in a given model \( \mathbb{M} \) the statement \( \text{Most}_M(A, B) \) says that \( \text{card}(A^M \cap B^M) > \text{card}(A^M - B^M) \), where \( A^M, B^M \subseteq M \).

2.2.5. Corollary. The definitions of a Lindström quantifier (2.2.1) and a generalized quantifier (2.2.4) are equivalent.
2.2. Generalized Quantifier Theory

Studying the properties of quantifiers in most cases we invoke Definition 2.2.1 but for descriptive purposes over some fixed universe we mostly make use of Definition 2.2.4, treating quantifiers as third-order concepts (relations between relations).

2.2.2 Branching Quantifiers

As a matter of chronology, the idea of generalizing Frege’s quantifiers arose much earlier than the work of Lindström (1966). The idea was to analyze possible dependencies between quantifiers — dependencies which are not allowed in the standard linear (Fregean) interpretation of logic. Branching quantification (also called partially ordered quantification, or Henkin quantification) was proposed by Leon Henkin (1961) (for a survey see Krynicki and Mostowski (1995)). Branching quantification significantly extends the expressibility of first-order logic; for example the so-called Ehrenfeucht sentence, which uses branching quantification, expresses infinity:

$$\exists t \left( \forall x \exists x' \left( \forall y \exists y' \left( (x = y \iff x' = y') \land x' \neq t \right) \right) \right).$$

Informally speaking, the idea of such a construction is that for different rows the values of the quantified variables are chosen independently. The semantics of branching quantifiers can be formulated mathematically in terms of Skolem functions. For instance, the Ehrenfeucht sentence after Skolemization has the following form:

$$\exists t \exists f \exists g \left[ \forall x \forall y \left( x = y \iff f(x) = g(y) \land f(x) \neq t \right) \right].$$

Via simple transformations this sentence is equivalent to the following:

$$\exists f \forall x \forall y \left[ (x \neq y \implies f(x) \neq f(y)) \land (\exists t \forall x (f(x) \neq t)) \right],$$

and therefore, it expresses Dedekind’s infinity: there exists an injective function from the universe to itself which is not surjective.

The idea of the independent (branching) interpretation of quantifiers has given rise to many advances in logic. Let us mention here only the logical study of (in)dependence by investigating Independence Friendly Logic (see Hintikka, 1996) and Dependence Logic (see Väänänen, 2007). It is also worth noting that Game-Theoretic Semantics (see Hintikka and Sandu, 1997) was originally designed as an alternative semantics for branching quantification (Independence Friendly Logic). Now it is considered as a useful tool for studying different variants of independence, like imperfect information games (see Sevenster, 2006). We present computational complexity results for branching quantifiers in Section 3.2 and discuss their linguistic applications in Chapter 6.
2.2.3 Logic Enriched by Generalized Quantifiers

Generalized quantifiers enable us to enrich the expressive power of a logic in a very controlled and minimal way. Below we define the extension of an arbitrary logic $\mathcal{L}$ by a generalized quantifier $Q$.

2.2.6. Definition. We define the extension, $\mathcal{L}(Q)$, of logic $\mathcal{L}$ by a quantifier $Q$ of type $t = (n_1, \ldots, n_k)$ in the following way:

- The formula formation rules of $\mathcal{L}$-language are extended by the rule:
  
  if for $1 \leq i \leq k$, $\varphi_i(\pi_i)$ is a formula and $\pi_i$ is an $n_i$-tuple of pairwise distinct variables, then $Q\pi_1, \ldots, \pi_k[\varphi_1(\pi_1), \ldots, \varphi_k(\pi_k)]$ is a formula.

2.2.7. Convention. Let us observe that this definition can be modified according to common notational habits as follows. $Q$ is treated as binding $n = \max(n_1, \ldots, n_k)$ variables in $k$ formulae. For example, the quantifier Every of type $(1,1)$ which expresses the property $\forall x[P_1(x) \implies P_2(x)]$ can be written according to the first convention as:

$$\text{Every } xy[P_1(x), P_2(y)]$$

and according to the modified one as:

$$\text{Every } x[P_1(x), P_2(x)].$$

2.2.8. Definition. The satisfaction relation of $\mathcal{L}$ is extended by the rule:

$$\mathcal{M} \models Q\pi_1, \ldots, \pi_k[\varphi_1(\pi_1), \ldots, \varphi_k(\pi_k)] \text{ iff } (M, \varphi_1^M, \ldots, \varphi_k^M) \in Q,$n$$

where $\varphi_i^M = \{\pi \in M^{n_i} | \mathcal{M} \models \varphi_i(\pi)\}$.

In this dissertation we will mainly be concerned with extensions of first-order logic, FO, by different generalized quantifiers $Q$. Following Definition 2.2.6 such an extension will be denoted by $\text{FO}(Q)$.

2.2.4 Definability of Generalized Quantifiers

Some generalized quantifiers, like $\exists^{\leq 3}$, $\exists^{=}^3$, and $\exists^{\geq 3}$, are easily expressible in elementary logic. This is also true for many natural language determiners. For example, we can express the type $(1,1)$ quantifier Some by the type $(1)$ first-order existential quantifier in the following way:

$$\text{Some } x[A(x), B(x)] \iff \exists x[A(x) \land B(x)].$$
However, it is well-known that many generalized quantifiers are not definable in first-order logic. Standard application of the compactness theorem shows that the quantifiers “there exist (in)finite many” are not FO-definable. Moreover, using Ehrenfeucht-Fraïssé games it is routine to prove the following:

2.2.9. Proposition. The quantifiers Most and Even are not first-order definable.\(^2\)

Dealing with quantifiers not definable in first-order logic we will consider their definitions in higher-order logics, for instance in fragments of second-order logic. To give one example here, in a model \(M = (M, A^M, B^M)\) the sentence Most \(x [A(x), B(x)]\) is true if and only if the following condition holds:

\[\exists f : (A^M - B^M) \rightarrow (A^M \cap B^M)\] such that \(f\) is injective but not surjective.

In general, definability theory investigates the expressibility of quantifiers in various logics. Informally, definability of a quantifier \(Q\) in a logic \(L\) means that there is a uniform way to express every formula of the form \(Q x \varphi\) in \(L\). The following is a precise definition.

2.2.10. Definition. Let \(Q\) be a generalized quantifier of type \(t\) and \(L\) a logic. We say that the quantifier \(Q\) is definable in \(L\) if there is a sentence \(\varphi \in L\) of vocabulary \(\tau_t\) such that for any \(\tau_t\)-structure \(M:\)

\[M \models \varphi \text{ iff } M \in Q.\]

2.2.11. Definition. Let \(L\) and \(L'\) be logics. The logic \(L'\) is at least as strong as the logic \(L\) \((L \leq L')\) if for every sentence \(\varphi \in L\) over any vocabulary there exists a sentence \(\psi \in L'\) over the same vocabulary such that:

\[\models \varphi \iff \psi.\]

The logics \(L\) and \(L'\) are equivalent \((L \equiv L')\) iff \(L \leq L'\) and \(L' \leq L\).

Below we assume that a logic \(L\) has the so-called substitution property, i.e., that the logic \(L\) is closed under substituting predicates by formulas. The following fact is well-known for Lindström quantifiers.

\(^2\)For more undefinability results together with mathematical details and an introduction into Ehrenfeucht-Fraïssé techniques we suggest consulting the literature (e.g. Chapter 4 in Peters and Westerståhl, 2006).
2.2.12. Proposition. Let $Q$ be a generalized quantifier and $L$ a logic. The quantifier $Q$ is definable in $L$ iff

$$L(Q) \equiv L.$$ 

Proof The direction $L \leq L(Q)$ is obvious since every formula of $L$ is also a formula of $L(Q)$ (see Definition 2.2.6). To show that $L(Q) \leq L$ we use inductively the fact that if $\varphi$ is the formula which defines $Q$ and $\psi_1(\overline{x_1}), \ldots, \psi_k(\overline{x_k})$ are formulae of $L$, then

$$|= Q(\overline{x_1}, \ldots, \overline{x_k})[\psi_1(\overline{x_1}), \ldots, \psi_k(\overline{x_k})] \iff \varphi(R_1/\psi_1, \ldots, R_k/\psi_k),$$

where the formula on the right is obtained by substituting every occurrence of $R_i(\overline{x}_i)$ in $\varphi$ by $\psi_i(\overline{x_i})$. □

2.2.5 Linguistic Properties of Generalized Quantifiers

It was realized by Montague (1970) that the notion of a generalized quantifier — as a relation between sets — can be used to model the denotations of noun phrases in natural language. Barwise and Cooper (1981) introduced the apparatus of generalized quantifiers as a standard semantic toolbox and started the rigorous study of their properties from the linguistic perspective. Below we define some properties of quantifiers important in linguistics which we use in our further research (see Peters and Westerståhl, 2006, for a general overview).

Boolean Combinations of Quantifiers

To account for complex noun phrases, like those occurring in sentences (7)–(10), we define disjunction, conjunction, outer negation (complement) and inner negation (post-complement) of generalized quantifiers.

(7) At least 5 or at most 10 departments can win EU grants. (disjunction)

(8) Between 100 and 200 students started in the marathon. (conjunction)

(9) Not all students passed. (outer negation)

(10) All students did not pass. (inner negation)

2.2.13. Definition. Let $Q$, $Q'$ be generalized quantifiers, both of type $(n_1, \ldots, n_k)$. We define:

$$(Q \land Q')_M[R_1, \ldots, R_k] \iff Q_M[R_1, \ldots, R_k] \land Q'_M[R_1, \ldots, R_k]$$ (conjunction)

$$(Q \lor Q')_M[R_1, \ldots, R_k] \iff Q_M[R_1, \ldots, R_k] \lor Q'_M[R_1, \ldots, R_k]$$ (disjunction).

$$(\neg Q)_M[R_1, \ldots, R_k] \iff \neg Q_M[R_1, \ldots, R_k]$$ (complement)

$$(Q \neg)_M[R_1, \ldots, R_k] \iff Q_M[R_1, \ldots, R_{k-1}, M - R_k]$$ (post-complement)
2.2. Generalized Quantifier Theory

Relativization of Quantifiers

Every statement involving a quantifier \( Q \) is about some universe \( M \). Sometimes it is useful to define a new quantifier saying that \( Q \) restricted to some subset of \( M \) behaves exactly as it behaves on the whole universe \( M \). Below we give the formal definition.

2.2.14. Definition. Let \( Q \) be of type \((n_1, \ldots, n_k)\); then the relativization of \( Q \), \( Q^{\text{rel}} \), has the type \((1, n_1, \ldots, n_k)\) and is defined for \( A \subseteq M, R_i \subseteq M^{n_i}, 1 \leq i \leq k \) as follows:

\[
Q^{\text{rel}}_M[A, R_1, \ldots, R_k] \iff Q_A[R_1 \cap A^{n_1}, \ldots, R_k \cap A^{n_k}].
\]

In particular, for \( Q \) of type \((1)\) we have:

\[
Q^{\text{rel}}_M[A, B] \iff Q_A[A \cap B].
\]

2.2.15. Example. This already shows that many natural language determiners of type \((1, 1)\) are just relativizations of some familiar logical quantifiers, e.g.:

\[
\begin{align*}
\text{Some} &= \exists^{\text{rel}}, \\
\text{Every} &= \forall^{\text{rel}}.
\end{align*}
\]

Domain Independence

Domain independence is a property characteristic of natural language quantifiers. It says that the behavior of a quantifier does not change when you extend the universe. The formal definition follows.

2.2.16. Definition. A quantifier of type \((n_1, \ldots, n_k)\) satisfies domain independence (EXT) iff the following holds:

If \( R_i \subseteq M^{n_i}, 1 \leq i \leq k, M \subseteq M' \), then \( Q_M[R_1, \ldots, R_k] \iff Q_{M'}[R_1, \ldots, R_k] \).

Conservativity

The property which in a sense extends EXT is conservativity:

2.2.17. Definition. A type \((1, 1)\) quantifier \( Q \) is conservative (CONS) iff for all \( M \) and all \( A, B \subseteq M \):

\[
Q_M[A, B] \iff Q_M[A, A \cap B].
\]
CE-quantifiers

Quantifiers closed under isomorphisms and satisfying CONS and EXT are known in the literature as CE-quantifiers. It has been hypothesized that all natural language determiners correspond to CE-quantifiers (see e.g. Barwise and Cooper, 1981). From this perspective expressions like “John”, ∀, and “only” are excluded from the realm of natural language determiners by not being CE-quantifiers. The proper name “John” does not satisfy isomorphism closure, ∀ is not EXT and “only” violates conservativity.

Notice that CE-quantifiers of type (1, 1) over finite universes may be identified with pairs of natural numbers.

2.2.18. DEFINITION. Let Q be a CE-quantifier of type (1, 1). We define a binary relation also denoted by Q:

\[ Q(k, m) \iff \text{there is } M, \text{ and } A, B \subseteq M \text{ such that } \text{card}(A - B) = k, \text{card}(A \cap B) = m, \text{ and } Q_M[A, B]. \]

2.2.19. THEOREM. If Q is a CE-quantifier of type (1, 1), then for all M and all A, B \( \subseteq M \) we have:

\[ Q_M[A, B] \iff Q(\text{card}(A - B), \text{card}(A \cap B)). \]

Proof It is enough to show that whenever Q is CE, A, B \( \subseteq M \), A', B' \( \subseteq M' \) are such that: \( \text{card}(A - B) = \text{card}(A' - B') \) and \( \text{card}(A \cap B) = \text{card}(A' \cap B') \), \( Q_M[A, B] \iff Q_{M'}[A', B'] \).

\[ \square \]

Monotonicity

One of the most striking intuitions about quantifiers is that they say that some sets are “large enough”. Therefore, we would expect that quantifiers are closed on some operations changing universe. The simplest among such operations is taking subsets and supersets. Monotonicity properties state whether a quantifier is closed under these operations.

2.2.20. DEFINITION. A quantifier Q_M of type \( (n_1, \ldots, n_k) \) is monotone increasing in the i’th argument (upward monotone) iff the following holds:

If Q_M[R_1, \ldots, R_k] and R_i \( \subseteq R'_i \subseteq M^{n_i} \), then
\[ Q_M[R_1, \ldots, R_{i-1}, R'_i, R_{i+1}, \ldots, R_k], \text{ where } 1 \leq i \leq k. \]

\[ \square \]
2.2. Generalized Quantifier Theory

2.2.21. Definition. A quantifier $Q_M$ of type $(n_1, \ldots, n_k)$ is monotone decreasing in the $i$'th argument (downward monotone) iff the following holds:

If $Q_M[R_1, \ldots, R_k]$ and $R'_i \subseteq R_i \subseteq M^{n_i}$, then

$$Q_M[R_1, \ldots, R_{i-1}, R'_i, R_{i+1}, \ldots, R_k],$$

where $1 \leq i \leq k$.

In particular, for a quantifier $Q$ of type $(1, 1)$ we can define the following basic types of monotonicity:

- $\uparrow\text{MON}$ $Q_M[A, B]$ and $A \subseteq A' \subseteq M$ then $Q_M[A', B]$.
- $\downarrow\text{MON}$ $Q_M[A, B]$ and $A' \subseteq A \subseteq M$ then $Q_M[A', B]$.
- MON↑ $Q_M[A, B]$ and $B \subseteq B' \subseteq M$ then $Q_M[A, B']$.
- MON↓ $Q_M[A, B]$ and $B' \subseteq B \subseteq M$ then $Q_M[A, B']$.

We also consider combinations of these basic types, for example we will write $\uparrow\text{MON}\downarrow$ for a quantifier that is monotone increasing in its left argument and decreasing in its right argument.

Upward monotonicity in the left argument is sometimes called persistence. It is an important property for the study of noun phrases in natural language (see e.g. Peters and Westerståhl, 2006).

2.2.22. Definition. We say that a quantifier is monotone if it is monotone decreasing or increasing in any of its arguments. Otherwise, we call it non-monotone.

Obviously, monotonicity interacts in a subtle way with outer and inner negations.

2.2.23. Proposition. Let $Q$ be any type $(1, 1)$ quantifier. $Q$ is MON↑

(1) iff $\neg Q$ is MON↓.

(2) iff $Q \neg$ is MON↓.

Q is ↑MON

(1) iff $\neg Q$ is ↓MON.

(2) iff $Q \neg$ is ↑MON.

Similarly (with reversed arrows) for the downward monotone case.
Chapter 2. Mathematical Prerequisites

Proof Obviously, outer negation reverses monotonicity, in any argument. Inner negation reverses monotonicity only in the second argument, since there we are looking at complements, but not in the first argument.

2.2.24. Example. Consider the Aristotelian square of opposition. It consists of the following four quantifiers: Some, ¬Some = No, Some ¬ = Not all, ¬Some ¬ = All. Some is ↑MON↑. Therefore, No is ↓MON↓, Not all is ↑MON↓, and All is ↓MON↑.

Moreover, Most is an example of a quantifier which is not persistent but is upward monotone in its right argument (i.e., ¬MON↑). Even is non-monotone (∼MON∼).

It is not surprising that monotonicity is one of the key properties of quantifiers, both in logic and linguistics. In model theory it contributes to definability (see e.g. Väänänen and Westerståhl, 2002), in linguistics it is used — among other applications — to explain the phenomenon of negative polarity items (see e.g. Ladusaw, 1979). Moreover, there are good reasons to believe that it is a crucial feature for processing natural language quantifiers, as has already been suggested by Barwise and Cooper (1981) and empirically supported by Geurts (2003) as well as our research presented in Chapter 7. There are also strong links between monotonicity and the learnability of quantifiers (see e.g. Tiede, 1999; Gierasimczuk, 2009).

2.3 Computability

Throughout the thesis we use the general notions of computability theory (see e.g. Hopcroft et al., 2000; Cooper, 2003). In particular, we refer to the basic methods and notions of complexity theory (see e.g. Papadimitriou, 1993; Kozen, 2006). Below we review some of them briefly to keep the thesis self-contained.

2.3.1 Languages and Automata

Formal language theory — which we briefly survey below — is an important part of logic, computer science and linguistics (see e.g. Hopcroft et al., 2000, for a complete treatment). Historically speaking, formal language theory forms the foundation of modern (mathematical) linguistics and its connection with psycholinguistics (see e.g. Partee et al., 1990).

Languages

By an alphabet we mean any non-empty finite set of symbols. For example, \( A = \{ a, b \} \) and \( B = \{ 0, 1 \} \) are two different binary alphabets.
A word (string) is a finite sequence of symbols from a given alphabet, e.g., “1110001110” is a word over the alphabet $B$.

The empty word is a sequence without symbols. It is needed mainly for technical reasons and written $\varepsilon$.

The length of a word is the number of symbols occurring in it. We write $lh()$ for length, e.g., $lh(111) = 3$ and $lh(\varepsilon) = 0$.

If $\Gamma$ is an alphabet, then by $\Gamma^k$ we mean the set of all words of length $k$ over $\Gamma$. For instance, $A^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$. For every alphabet $\Gamma$ we have $\Gamma^0 = \{\varepsilon\}$.

For any letter $a$ and a natural number $n$ by $a^n$ we denote a string of length $n$ consisting only from the letter $a$.

The set of all words over alphabet $\Gamma$ is denoted by $\Gamma^*$, e.g., $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. In other words, $\Gamma^* = \bigcup_{n \in \omega} \Gamma^n$. Almost always $\Gamma^*$ is infinite, except for two cases: for $\Gamma = \emptyset$ and $\Gamma = \{\varepsilon\}$.

By $xy$ we mean the concatenation of the word $x$ with the word $y$, i.e., the new word $xy$ is built from $x$ followed by $y$. If $x = a_1 \ldots a_i$ and $y = b_1 \ldots b_n$, then $xy$ is of length $i + n$ and $xy = a_1 \ldots a_i b_1 \ldots b_n$. For instance, if $x = 101$ and $y = 00$, then $xy = 10100$. For any string $\alpha$ the following holds: $\varepsilon \alpha = \alpha \varepsilon = \alpha$. Hence, $\varepsilon$ is the neutral element for concatenation.

Any set of words, a subset of $\Gamma^*$, will be called a language. If $\Gamma$ is an alphabet and $L \subseteq \Gamma^*$, then we say that $L$ is a language over $\Gamma$. For instance, the set $L \subseteq A^*$ such that $L = \{\alpha \mid \text{the number of occurrences of } b \text{ in } \alpha \text{ is even}\}$ is a language over the alphabet $A$.

**Finite Automata**

A finite state automaton is a model of computation consisting of a finite number of states and transitions between those states. We give a formal definition below.

**2.3.1. Definition.** A **non-deterministic finite automaton** (FA) is a tuple $(A, Q, q_s, F, \delta)$, where:

- $A$ is an input alphabet;
- $Q$ is a finite set of states;
- $q_s \in Q$ is an initial state;
- $F \subseteq Q$ is a set of accepting states;
- $\delta : Q \times A \rightarrow \mathcal{P}(Q)$ is a transition function.
If $H = (A, Q, q_s, \delta, F)$ is a FA such that for every $a \in A$ and $q \in Q$ we have $\text{card}(\delta(q, a)) \leq 1$, then $H$ is a deterministic automaton. In that case we can describe a transition function as a partial function: $\delta : Q \times A \rightarrow Q$.

Finite automata are often presented as graphs, where vertices (circles) symbolize internal states, the initial state is marked by an arrow, an accepting state is double circled, and arrows between nodes describe a transition function on letters given by the labels of these arrows. We will give a few examples in what follows.

2.3.2. Definition. Let us first define the generalized transition function $\bar{\delta}$ which describes the behavior of an automata reading a string $w$ from the initial state $q$:

$$\bar{\delta} : Q \times A^* \rightarrow \mathcal{P}(Q),$$

where:

$$\bar{\delta}(q, \varepsilon) = \{q\}$$

and for each $w \in A^*$ and $a \in A$, $\bar{\delta}(q, wa) = \bigcup_{q' \in \bar{\delta}(q, w)} \delta(q', a)$.

2.3.3. Definition. The language accepted (recognized) by some FA $H$ is the set of all words over the alphabet $A$ which are accepted by $H$, that is:

$$L(H) = \{w \in A^* : \bar{\delta}(q_s, w) \cap F \neq \emptyset\}.$$  

2.3.4. Definition. We say that a language $L \subseteq A^*$ is regular if and only if there exists some FA $H$ such that $L = L(H)$.

The following equivalence is a well known fact.

2.3.5. Theorem. Deterministic and non-deterministic finite automata recognize the same class of languages, i.e. regular languages.

Proof First of all notice that every deterministic FA is a non-deterministic FA. Then we have to only show that every non-deterministic FA can be simulated by some deterministic FA. The proof goes through the so-called subset construction. It involves constructing all subsets of the set of states of the non-deterministic FA and using them as states of a new, deterministic FA. The new transition function is defined naturally (see Hopcroft et al., 2000, for details). Notice that in the worst case the new deterministic automaton can have $2^n$ states, where $n$ is the number of states of the corresponding non-deterministic automaton.
2.3. Computation

2.3.6. Example. Let us give a few simple examples of regular languages together with the corresponding accepting automata.

Let \( A = \{a, b\} \) and consider the language \( L_1 = A^* \). \( L_1 = L(H_1) \), where
\[
H_1 = (Q_1, q_1, F_1, \delta_1) \text{ such that: } Q_1 = \{q_1\}, F_1 = \{q_1\}, \delta_1(q_1, a) = q_1 \text{ and } \delta_1(q_1, b) = q_1. \]
The automaton is shown in Figure 2.1.

\[
\begin{array}{c}
a, b \\
\rightarrow \\
q_1
\end{array}
\]

Figure 2.1: Finite automaton recognizing language \( L_1 = A^* \).

Now let \( L_2 = \emptyset \); then \( L_2 = L(H_2) \), where
\[
H_2 = (Q_2, q_2, F_2, \delta_2) \text{ such that: } Q_2 = \{q_2\}, F_2 = \emptyset, \delta_2(q_2, a) = q_2 \text{ and } \delta_2(q_2, b) = q_2. \]
The automaton is depicted in Figure 2.2.

\[
\begin{array}{c}
a, b \\
\rightarrow \\
q_2
\end{array}
\]

Figure 2.2: Finite automaton recognizing language \( L_2 = \emptyset \).

Finally, let \( L_3 = \{\varepsilon\} \). \( L_3 = L(H_3) \), where
\[
H_3 = (Q_3, q_0, F_3, \delta_3) \text{ such that: } Q_3 = \{q_0, q_1\}, F_3 = \{q_0\}, \delta_3(q_0, i) = q_1 \text{ and } \delta_3(q_1, i) = q_1, \text{ for } i = a, b.
\]
The finite automaton accepting this language is presented in Figure 2.3.

\[
\begin{array}{c}
a, b \\
\rightarrow \\
q_0 \\
\rightarrow \\
q_1
\end{array}
\]

Figure 2.3: Finite automaton recognizing language \( L_3 = \{\varepsilon\} \).

Beyond Finite Automata

It is well-known that not every formal language is regular, i.e., recognized by a finite automata. For example, the language \( L_{ab} = \{a^n b^n : n \geq 1\} \) cannot be
recognized by any finite automaton. Why is that? Strings from that language can be arbitrary long and to recognize them a machine needs to count whether the number of letters “a” is equal to the number of letters “b”. A string from $L_{ab}$ can start with any number of letters “a” so the corresponding machine needs to be able to memorize an arbitrarily large natural number. To do this a machine has to be equipped with an unbounded internal memory. However, a finite automaton with $k$ states can remember only numbers smaller than $k$. This claim is precisely formulated in the following lemma which implies that the language $L_{ab}$ is not regular.

2.3.7. Theorem (Pumping Lemma for Regular Languages). For any infinite regular language $L \subseteq A^*$ there exists a natural number $n$ such that for every word $\alpha \in L$, if $lh(\alpha) \geq n$, then there are $x, y, z \in A^*$ such that:

(1) $\alpha = xyz$;
(2) $y \neq \varepsilon$;
(3) $lh(xz) \leq n$;
(4) For every $k \geq 0$ the string $xy^kz$ is in $L$.

Push-down Automata

To account for languages which are not regular we need to extend the concept of a finite automaton. A push-down automaton (PDA) is a finite automaton that can make use of a stack (internal memory). The definition follows.

2.3.8. Definition. A non-deterministic push-down automaton (PDA) is a tuple $(A, \Gamma, \#, Q, q_s, F, \delta)$, where:

- $A$ is an input alphabet;
- $\Gamma$ is a stack alphabet;
- $\# \notin \Gamma$ is a stack initial symbol, empty stack consists only from it;
- $Q$ is a finite set of states;
- $q_s \in Q$ is an initial state;
- $F \subseteq Q$ is a set of accepting states;
- $\delta : Q \times (A \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma^*)$ is a transition function. We denote a single transition by: $(q, a, n) \xrightarrow{H} (p, \gamma)$, if $(p, \gamma) \in \delta(q, a, n)$, where $q, p \in Q, a \in A, n \in \Gamma, \gamma \in \Gamma^*$. 

\[ \blacksquare \]
2.3. Computability

If $H = (A, \Gamma, \#, Q, q_s, q_a, \delta)$ is a PDA and for every $a \in A$, $q \in Q$ and $\gamma \in \Gamma \text{ card}(\delta(q, a, \gamma)) \leq 1$ and $\delta(q, \varepsilon, \gamma) = \emptyset$, then $H$ is a deterministic push-down automaton (DPDA).

The language recognized by a PDA $H$ is the set of strings accepted by $H$. A string $w$ is accepted by $H$ if and only if starting in the initial state $q_0$ with the empty stack and reading the string $w$, the automaton $H$ terminates in an accepting state $p \in F$.

2.3.9. Definition. We say that a language $L \subseteq A^*$ is context-free if and only if there is a PDA $H$ such that $L = L(H)$.

Observe that (non-deterministic) PDAs accept a larger class of languages than DPDAs. For instance, the language consisting of palindromes is context-free but cannot be recognized by any DPDA as a machine needs to "guess" which is the middle letter of every string.

2.3.10. Example. Obviously, the class of all context-free languages is larger than the class of all regular languages. For instance, the language $L_{ab} = \{a^n b^n : n \geq 1\}$, which we argued to be non-regular, is context-free. To show this we will construct a PDA $H$ such that $L_{ab} = L(H)$. $H$ recognizes $L_{ab}$ reading every string from left to right and pushes every occurrences of the letter "$a$" to the top of the stack. After finding the first occurrence of the letter "$b$" the automaton $H$ pops an "$a$" off the stack when reading each "$b$". $H$ accepts a string if after processing all of it the stack is empty.

Formally, let $H = (A, \Gamma, \#, Q, q_s, q_a, \delta)$, where $A = \{a, b\} = \Gamma$, $Q = \{q_s, q_1, q_2, q_a\}$, $F = \{q_a\}$ and the transition function is specified in the following way:

- $(q_s, a, \#) \xrightarrow{H} (q_s, \#a)$;
- $(q_s, a, a) \xrightarrow{H} (q_s, aa)$;
- $(q_s, b, a) \xrightarrow{H} (q_1, \varepsilon)$;
- $(q_s, b, \#) \xrightarrow{H} (q_2, \varepsilon)$;
- $(q_1, \varepsilon, \#) \xrightarrow{H} (q_a, \varepsilon)$;
- $(q_1, b, a) \xrightarrow{H} (q_1, \varepsilon)$;
- $(q_1, b, b) \xrightarrow{H} (q_2, \varepsilon)$;
- $(q_1, a, b) \xrightarrow{H} (q_2, \varepsilon)$;
Chapter 2. Mathematical Prerequisites

• \((q_1, b, \#) \xrightarrow{H} (q_2, \varepsilon);\)

• \((q_1, a, \#) \xrightarrow{H} (q_2, \varepsilon).\)

Context-free languages also have restricted description power. For example, the language \(L_{abc} = \{a^k b^k c^k : k \geq 1\}\) is not context-free. This fact follows from the extended version of the pumping lemma.

2.3.11. Theorem (Pumping Lemma for Context-free Languages).
For every context-free language \(L \subseteq A^*\) there is a natural number \(k\) such that for each \(w \in L\), if \(lh(w) \geq k\), then there are \(\beta_1, \beta_2, \gamma_1, \gamma_2, \eta\) such that:

• \(\gamma_1 \neq \varepsilon\) or \(\gamma_2 \neq \varepsilon;\)

• \(w = \beta_1 \gamma_1 \eta \gamma_2 \beta_2;\)

• for every \(m \in \omega: \beta_1 \gamma_1^m \eta \gamma_2^m \beta_2 \in L.\)

Extending push-down automata with more memory (e.g., one additional stack) we reach the realm of Turing machines.

2.3.2 Turing Machines

The basic device of computation in this thesis is a multi-tape Turing (1936) machine. Most of the particulars of Turing machines are not of direct interest to us. Nevertheless, we recall the basic idea. A multi-tape Turing machine consists of a read-only input tape, a read and write working tape, and a write-only output tape. Every tape is divided into cells scanned by the read-write head of the machine. Each cell contains a symbol from some finite alphabet. The tapes are assumed to be arbitrarily extendable to the right. At any time the machine is in one of a finite number of states. The actions of a Turing machine are determined by a finite programme which determines, according to the current configuration (i.e., the state of the machine and the symbols in the cells being scanned) which action should be executed next. A computation of a Turing machine consists thus of a series of successive configurations. A Turing machine is deterministic if its state transitions are uniquely defined, otherwise it is non-deterministic. Therefore, a deterministic Turing machine has a single computation path (for any particular input) and a non-deterministic Turing machine has a computation tree. A Turing machine accepts an input if its computation on that inputs halts after finite time in an accepting state. It rejects an input if it halts in a rejecting state.

2.3.12. Definition. Let \(\Gamma\) be some finite alphabet and \(L \subseteq \Gamma^*\) a language. We say that a deterministic Turing machine, \(M\), decides \(L\) if for every \(x \in \Gamma^*\) \(M\) halts in the accepting state on \(x\) whenever \(x \in L\) and in the rejecting state, otherwise.
A non-deterministic Turing machine, $M$, recognizes $L$ if for every $x \in L$ there is a computation of $M$ which halts in the accepting state and there is no such computation for any $x \notin L$.

It is important to notice that non-deterministic Turing machines recognize the same class of languages as deterministic ones. This means that for every problem which can be recognized by a non-deterministic Turing machine there exists a deterministic Turing machine deciding it.

2.3.13. Theorem. If there is a non-deterministic Turing machine $N$ recognizing a language $L$, then there exists a deterministic Turing machine $M$ for language $L$.

Proof The basic idea for simulating $N$ is as follows. Machine $M$ considers all computation paths of $N$ and simulates $N$ on each of them. If $N$ would halt on a given computation path in an accepting state then $M$ also accepts. Otherwise, $M$ moves to consider the next computation path of $N$. $M$ rejects the input if machine $N$ would not halt in an accepting state at any computation path.

However, the length of an accepting computation of the deterministic Turing machine is, in general, exponential in the length of the shortest accepting computation of the non-deterministic Turing machines as a deterministic machine has to simulate all possible computation paths of a non-deterministic machine. The question whether this simulation can be done without exponential growth in computation time leads us to computational complexity theory.

2.3.3 Complexity Classes

Let us start our complexity considerations with the notation used for comparing the growth rates of functions.

2.3.14. Definition. Let $f, g : \omega \rightarrow \omega$ be any functions. We say that $f = O(g)$ if there exists a constant $c > 0$ such that $f(n) \leq cg(n)$ for almost all (i.e., all but finitely many) $n$.

Let $f : \omega \rightarrow \omega$ be a natural number function. TIME($f$) is the class of languages (problems) which can be recognized by a deterministic Turing machine in time bounded by $f$ with respect to the length of the input. In other words,

\[\text{TIME}(f) = \{ L \mid \exists M \text{ Turing machine, } L \in \text{Accept}(M) \land T_M(n) \leq f(n) \} \]
Chapter 2. Mathematical Prerequisites

$L \in \text{TIME}(f)$ if there exists a deterministic Turing machine such that for every $x \in L$, the computation path of $M$ on $x$ is shorter than $f(n)$, where $n$ is the length of $x$. \text{TIME}(f)$ is called a deterministic computational complexity class. A non-deterministic complexity class, \text{NTIME}(f), is the class of languages $L$ for which there exists a non-deterministic Turing machine $M$ such that for every $x \in L$ all branches in the computation tree of $M$ on $x$ are bounded by $f(n)$ and moreover $M$ decides $L$. One way of thinking about a non-deterministic Turing machine bounded by $f$ is that it first guesses the right answer and then deterministically in a time bounded by $f$ checks if the guess is correct.

\text{SPACE}(f)$ is the class of languages which can be recognized by a deterministic machine using at most $f(n)$ cells of the working-tape. \text{NSPACE}(f)$ is defined analogously.

Below we define the most important and well-known complexity classes, i.e., the sets of languages of related complexity. In other words, we can say that a complexity class is the set of problems that can be solved by a Turing machine using $O(f(n))$ of time or space resource, where $n$ is the size of the input. To estimate these resources mathematically natural functions have been chosen, like logarithmic, polynomial and exponential functions. It is well known that polynomial functions grow faster than any logarithmic functions and exponential functions dominate polynomial functions. Therefore, it is commonly believed that problems belonging to logarithmic classes need essentially less resources to be solved than problems from the polynomial classes and likewise that polynomial problems are easier than exponential problems.

2.3.15. Definition.

- $\text{LOGSPACE} = \bigcup_{k \in \omega} \text{SPACE}(k \log n)$
- $\text{NLOGSPACE} = \bigcup_{k \in \omega} \text{NSPACE}(k \log n)$
- $\text{PTIME} = \bigcup_{k \in \omega} \text{TIME}(n^k)$
- $\text{NP} = \bigcup_{k \in \omega} \text{NTIME}(n^k)$
- $\text{PSPACE} = \bigcup_{k \in \omega} \text{SPACE}(n^k)$
- $\text{NPSPACE} = \bigcup_{k \in \omega} \text{NSPACE}(n^k)$
- $\text{EXPTIME} = \bigcup_{k \in \omega} \text{TIME}(k^n)$
- $\text{NEXPTIME} = \bigcup_{k \in \omega} \text{NTIME}(k^n)$
If \( L \in \text{NP} \), then we say that \( L \) is decidable (computable, solvable) in non-deterministic polynomial time and likewise for other complexity classes.

It is obvious that for any pair of the complexity classes presented above, the lower one includes the upper one. However, when it comes to the strictness of these inclusions not much is known. One instance that has been proven is for \( \text{LOGSPACE} \) and \( \text{PSPACE} \) (see e.g. Papadimitriou, 1993, for so-called Hierarchy Theorems).

The complexity class of all regular languages, i.e., languages recognized by finite automata, is sometimes referred to as \( \text{REG} \) and equals \( \text{SPACE}(O(1)) \), the decision problems that can be solved in constant space (the space used is independent of the input size). The complexity class of all languages recognized by push-down automata (i.e. context-free languages) is contained in \( \text{LOGSPACE} \).

The question whether \( \text{PTIME} \) is strictly contained in \( \text{NP} \) is the famous Millennium Problem — one of the most fundamental problems in theoretical computer science, and in mathematics in general. The importance of this problem reaches well outside the theoretical sciences as the problems in \( \text{NP} \) are usually taken to be intractable or not efficiently computable as opposed to the problems in \( \text{P} \) which are conceived of as efficiently solvable. In the thesis we take this distinction for granted and investigate semantic constructions in natural language from that perspective (see Chapter 1 for a discussion of this claim).

Moreover, it was shown by Walter Savitch (1970) that if a nondeterministic Turing machine can solve a problem using \( f(n) \) space, an ordinary deterministic Turing machine can solve the same problem in the square of the space. Although it seems that nondeterminism may produce exponential gains in time, this theorem shows that it has a markedly more limited effect on space requirements.

**2.3.16. Theorem (Savitch (1970)).** For any function \( f(n) \geq \log(n) \):

\[
\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f(n)^2).
\]

**2.3.17. Corollary.** \( \text{PSPACE} = \text{NPSPACE} \)

**2.3.18. Definition.** For any computation class \( C \) we will denote by \( \text{co-C} \) the class of complements of languages in \( C \).

Every deterministic complexity class coincides with its complement. It is enough to change accepting states into rejecting states to get a machine computing the complement \( L \) from a deterministic machine deciding \( L \) itself. However, it is unknown whether \( \text{NP} = \text{co-NP} \). This is very important questions, as \( \text{P} = \text{NP} \) would imply that \( \text{NP} = \text{co-NP} \).

**2.3.4 Oracle Machines**

An oracle machine can be described as a Turing machine with a black box, called an oracle, which is able to decide certain decision problems in a single step. More
precisely, an oracle machine has a separate write-only oracle tape for writing down queries for the oracle. In a single step, the oracle computes the query, erases its input, and writes its output to the tape.

2.3.19. Definition. If \( B \) and \( C \) are complexity classes, then \( B \) relativized to \( C \), \( B^C \), is the class of languages recognized by oracle machines which obey the bounds defining \( B \) and use an oracle for problems belonging to \( C \).

2.3.5 The Polynomial Hierarchy

The Polynomial Hierarchy, \( PH \), is a very well-known hierarchy of classes above \( NP \). It is usually defined inductively using oracle machines and relativization (see e.g. Papadimitriou, 1993) as below.

2.3.20. Definition.

\[
\begin{align*}
1. & \quad \Sigma_1^P = NP; \\
2. & \quad \Sigma_{n+1}^P = NP^{\Sigma_n^P}; \\
3. & \quad \Pi_n^P = \text{co-}\Sigma_n^P; \\
4. & \quad PH = \bigcup_{i \geq 1} \Sigma_i^P.
\end{align*}
\]

It is known that \( PH \subseteq \text{PSPACE} \) (see e.g. Papadimitriou, 1993).

2.3.6 The Counting Hierarchy

The polynomial hierarchy defined above is an oracle hierarchy with \( NP \) as the building block. If we replace \( NP \) by probabilistic polynomial time, \( PP \), in the definition of \( PH \), then we arrive at a class called the counting hierarchy, \( CH \). The class \( PP \) consists of languages \( L \) for which there is a polynomial time bounded nondeterministic Turing machine \( M \) such that, for all inputs \( x \), \( x \in L \) iff more than half of the computations of \( M \) on input \( x \) end up accepting. In other words, a language \( L \) belongs to the class \( PP \) iff it is accepted with a probability more than one-half by some nondeterministic Turing machine in polynomial time.

The counting hierarchy can be defined now as follows, in terms of oracle Turing machines:

2.3.21. Definition.

\[
\begin{align*}
1. & \quad C_0^P = \text{PTIME}; \\
2. & \quad C_{k+1}^P = \text{PP}^{C_k^P};
\end{align*}
\]
2.3. Computability

\[ \text{CH} = \bigcup_{k \in \mathbb{N}} C_k P. \]

It is known that the polynomial hierarchy, \( \text{PH} \), is contained in the second level \( C_2 P \) of the counting hierarchy \( \text{CH} \) (see Toda, 1991). The question whether \( \text{CH} \subseteq \text{PH} \) is still open. However, it is widely believed that this is not the case. Under this assumption we will deliver in Chapter 5 an argument for restrictions on the type-shifting strategy in modeling the meanings of collective determiners in natural language.

2.3.7 Reductions and Complete Problems

The intuition that some problems are more difficult than others is formalized in complexity theory by the notion of a reduction. We will use only polynomial time many-one (Karp (1972)) reductions.

2.3.22. Definition. We say that a function \( f : A \rightarrow A \) is a polynomial time computable function iff there exists a deterministic Turing machine computing \( f(w) \) for every \( w \in A \) in polynomial time.

2.3.23. Definition. A problem \( L \subseteq \Gamma^* \) is polynomial reducible to a problem \( L' \subseteq \Gamma^* \) if there is a polynomial time computable function \( f : \Gamma^* \rightarrow \Gamma^* \) from strings to strings, such that

\[ w \in L \iff f(w) \in L'. \]

We will call such function \( f \) a polynomial time reduction of \( L \) to \( L' \).

2.3.24. Definition. A language \( L \) is complete for a complexity class \( \mathcal{C} \) if \( L \in \mathcal{C} \) and every language in \( \mathcal{C} \) is reducible to \( L \).

Intuitively, if \( L \) is complete for a complexity class \( \mathcal{C} \) then it is among the hardest problems in this class. The theory of complete problems was initiated with a seminal result of Cook (1971), who proved that the satisfiability problem for propositional formulae, \( \text{SAT} \), is complete for \( \text{NP} \). Many other now famous problems were then proven \( \text{NP} \)-complete by Karp (1972) — including some versions of satisfiability, like \( 3\text{SAT} \) (the restriction of \( \text{SAT} \) to formulae in conjunctional normal form such that every clause contains 3 literals), as well as some graph problems, e.g. \( \text{CLIQUE} \), which we define below. The book of Garey and Johnson (1979) contains a list of \( \text{NP} \)-complete languages.

2.3.25. Example. Let us give an example of a polynomial reduction. We will prove that the problem \( \text{CLIQUE} \) is \( \text{NP} \)-complete by reducing \( 3\text{SAT} \) to it. We will define other versions of the \( \text{CLIQUE} \) problem and use them to prove some complexity results for quantifiers in Chapter 3.
2.3.26. Definition. Let \( G = (V, E) \) be a graph and take a set \( C \subseteq V \). We say that \( C \) is a clique if there is \((i, j) \in E\) for every \( i, j \in C \). 

2.3.27. Definition. The problem clique can be formulated now as follows. Given a graph \( G = (V, E) \) and a natural number \( k \), determine whether there is a clique in \( G \) of cardinality at least \( k \).

2.3.28. Theorem. clique is NP-complete.

Proof First we have to show that clique belongs to NP. Once we have located \( k \) or more vertices which form a clique, it is trivial to verify that they do, this is why the clique problem is in NP.

To show NP-hardness we will reduce \( 3\text{SAT} \) to clique. Assume that our input is a set of clauses in the form of \( 3\text{SAT} \):
\[
Z = \{ (\ell_1^1 \lor \ell_2^1 \lor \ell_3^1), \ldots, (\ell_1^m \lor \ell_2^m \lor \ell_3^m) \},
\]
where \( \ell_i^j \) is a literal. We construct \((G, k)\) such that:

- \( k = m \);
- \( G = (V, E) \), where:

  - \( V = \{ v_{ij} \mid i = 1, \ldots, m; j = 1, 2, 3 \} \);
  - \( E = \{ (v_{ij}, v_{lk}) \mid i \neq l; \ell_i^j \neq \neg \ell_k^l \} \).

To complete the proof it suffices to observe that in graph \( G \) there is a clique of cardinality \( k \) if and only if the set \( Z \) is satisfiable (see e.g. Papadimitriou, 1993). □

2.4 Descriptive Complexity Theory

Classical descriptive complexity deals with the relationship between logical definability and computational complexity. The main idea is to treat classes of finite models over a fixed vocabulary as computational problems. In such a setting rather than the computational complexity of a given class of models we are dealing with its descriptive complexity, i.e., the question how difficult it is to describe the class using some logic. This section very briefly explains the fundamentals of descriptive complexity as a subfield of finite model theory. More details may be found in the books of Immerman (1998), Ebbinghaus and Flum (2005), Libkin (2004), and Grädel et al. (2007).
2.4.1 Encoding Finite Models

$M$ is a finite model when its universe, $M$, is finite. A widely studied class of finite models are graphs, i.e., structures of the form $G = (V, E)$, where the finite universe $V$ is called the set of vertices (nodes) of the graph and a binary relation $E \subseteq V^2$ is the set of edges of the graph.

Notice that in a linguistic context it makes sense to restrict ourselves to finite models as in a typical communication situation we refer to relatively small finite sets of objects. We have discussed this assumption further in Section 1.6.

Let $K$ be a class of finite models over some fixed vocabulary $\tau$. We want to treat $K$ as a problem (language) over the vocabulary $\tau$. To do this we need to code $\tau$-models as finite strings. We can assume that the universe of a model $M \in K$ consists of natural numbers: $M = \{1, \ldots, n\}$. A natural way of encoding a model $M$ (up to isomorphism) is by listing its universe, $M$, and storing the interpretation of the symbols in $\tau$ by writing down their truth-values on all tuples of objects from $M$.

2.4.1. Definition. Let $\tau = \{R_1, \ldots, R_k\}$ be a relational vocabulary and $M$ a $\tau$-model of the following form: $M = (M, R_1, \ldots, R_k)$, where $M = \{1, \ldots, n\}$ is the universe of model $M$ and $R_i \subseteq M$ is an $n_i$-ary relation over $M$, for $1 \leq i \leq k$. We define a binary encoding for $\tau$-models. The code for $M$ is a word over $\{0, 1, \#\}$ of length $O((\text{card}(M))^c)$, where $c$ is the maximal arity of the predicates in $\tau$ (or $c = 1$ if there are no predicates).

The code has the following form:

\[ \tilde{n}\#\tilde{R}_1\#\ldots\#\tilde{R}_n, \]

where:

- $\tilde{n}$ is the part coding the universe of the model and consists of $n$ 1s.
- $\tilde{R}_i$ — the code for the $n_i$-ary relation $R_i$ — is an $n^i$-bit string whose $j$-th bit is 1 iff the $j$-th tuple in $M^{n_i}$ (ordered lexicographically) is in $R_i$.
- $\#$ is a separating symbol.\(^4\)

2.4.2. Example. Let us give an example of a binary code corresponding to a model. Consider vocabulary $\sigma = \{P, R\}$, where $P$ is a unary predicate and $R$ a binary relation. Take the $\sigma$-model $M = (M, P^M, R^M)$, where the universe $M = \{1, 2, 3\}$, the unary relation $P^M \subseteq M$ is equal to $\{2\}$ and the binary relation $R^M \subseteq M^2$ consists of the pairs $(2, 2)$ and $(3, 2)$. Notice, that we can think about such models as graphs in which some nodes are “colored” by $P$.

Let us construct the code step by step:

\[^4\]See also Definition 2.1 of (Immerman, 1998) for a binary coding without separating symbol.
Chapter 2. Mathematical Prerequisites

- \( \tilde{n} \) consists of three 1s as there are three elements in \( M \).

- \( P^M \) is the string of length three with 1s in places corresponding to the elements from \( M \) belonging to \( P^M \). Hence \( P^M = 010 \) as \( P^M = \{2\} \).

- \( \tilde{R}^M \) is obtained by writing down all \( 3^2 = 9 \) binary strings of elements from \( M \) in lexicographical order and substituting 1 in places corresponding to the pairs belonging to \( R^M \) and 0 in all other places. As a result \( R^M = 000010010 \).

Adding all together the code for \( M \) is 111#010#000010010.

2.4.2 Logics Capturing Complexity Classes

Now we can formulate the central definition of descriptive complexity theory.

2.4.3. Definition. Let \( L \) be a logic and \( C \) a complexity class. We say that \( L \) captures \( C \), if for any vocabulary \( \tau \) and any class of \( \tau \)-models the following holds:

\[ \mathcal{K} \text{ is in } C \text{ if and only if } \mathcal{K} \text{ is definable by an } L\text{-sentence.} \]

The following are two classical results of descriptive complexity theory:

2.4.4. Theorem (Fagin (1974)). \( \Sigma_1^1 \) captures \( NP \).

2.4.5. Theorem (Stockmeyer (1976)). For any \( m \), \( \Sigma_m^1 \) captures \( \Sigma_m^P \).

Fagin’s theorem establishes a correspondence between existential second order logic and \( NP \). Stockmeyer’s extends it for the hierarchy of second-order formulae and the polynomial hierarchy. There are many other logical characterizations of complexity classes known (see e.g. Immerman, 1998), for instance that first-order logic is contained in \( \text{LOGSPACE} \) (see Immerman, 1998, Theorem 3.1). One of the famous results is the characterization of \( \text{PTIME} \) over ordered graph structures in terms of fixed-point logic, due to Immerman (1982) and Vardi (1982). Namely, in the presence of a linear ordering of the universe it is possible to use tuples of nodes to build a model of a Turing machine inside the graph and imitate the polynomial time property by a suitable fixed point sentence (e.g. see Immerman, 1998). One of the most important open problems is the question what logic \( L \) captures \( \text{PTIME} \) on graphs if we do not have an ordering of the vertices. Knowing \( L \) one could try to show that \( L \neq \Sigma_1^1 \), from which it would follow \( P \neq NP \).

Kontinen and Niemistö (2006) showed that the extension of first-order logic by second-order majority quantifiers of all arities (see Section 5.3 for a definition)
describes exactly problems in the counting hierarchy. We will investigate the linguistic consequences of that claim in Chapter 5.

Let us now define one of the crucial concepts of descriptive complexity which we use throughout the thesis.

2.4.6. Definition. If every $\mathcal{L}$-definable class $\mathcal{K}$ is in $\mathcal{C}$ then we say that model checking (data complexity) for $\mathcal{L}$ is in $\mathcal{C}$.

2.4.7. Remark. In the computational literature many other complexity measures besides model-checking are considered, most notably the so-called expression complexity and combined complexity introduced by Vardi (1982). The main difference between them and model-checking is as follows. In the latter our input is a model and we measure complexity with respect to the size of its universe. For expression complexity a formula from some set is fixed as an input and we measure its complexity — given as a function of the length of the expression — in different models. Expression complexity is sometimes referred to as a measure for succinctness of a language. There is a great difference between those two measures, for example $\Sigma_1^1$ expression complexity is NEXPTIME, but its model-checking is NP-complete (see e.g. Gottlob et al., 1999, for a systematic comparison). The third possibility is given by combined complexity: both a formula and a structure are given as an input and the complexity is defined with respect to their combined size. In this thesis we investigate model-checking complexity for quantifiers.

2.5 Quantifiers in Finite Models

Below we review some recent work in the field of generalized quantifiers on finite models. For more detailed discussion of this subject we refer to the survey of Makowsky and Pnueli (1995) and the paper of Väänänen (1997a).

Recall Definition 2.2.1, which says that generalized quantifiers are simply classes of models. Finite models can be encoded as finite strings over some vocabulary (recall Definition 2.4.1). Therefore, generalized quantifiers can be treated as classes of such finite strings, i.e., languages. Now we can easily fit the notions into the descriptive complexity paradigm.

2.5.1. Definition. By the complexity of a quantifier $Q$ we mean the computational complexity of the corresponding class of finite models.

2.5.2. Example. Consider a quantifier of type $(1, 2)$: a class of finite colored graphs of the form $M = (M, A^M, R^M)$. Let us take a model of this form, $M$, and a quantifier $Q$. Our computational problem is to decide whether $M \in Q$; or equivalently, to solve the query whether $M \models Q[A, R]$. 
This can simply be viewed as the model-checking problem for quantifiers. These notions can easily be generalized to quantifiers of arbitrary types \((n_1, \ldots, n_k)\) by considering classes of models of the form \(M = (M, R_1, \ldots, R_k)\), where \(R_i \subseteq M^{n_i}\), for \(i = 1, \ldots, k\).

Generalized quantifiers in finite models were considered from the point of view of computational complexity for the first time by Blass and Gurevich (1986). They investigated various forms of branching (Henkin) quantifiers (see Section 2.2.2) defining NP or NLOGSPACE complete graph problems. They introduced the following terminology.

2.5.3. Definition. We say that quantifier \(Q\) is NP-hard if the corresponding class of finite models is NP-hard. \(Q\) is mighty (NP-complete) if the corresponding class belongs to NP and is NP-hard.

In the previous chapter we mentioned that one of the fundamental problems of descriptive complexity theory is to find a logic which expresses exactly the polynomial time queries on unordered structures. Studying the computational complexity of quantifiers can contribute to this question. For instance, Hella et al. (1996) have proven that there is a representation of PTIME queries in terms of fixed-point logic enriched by the quantifier \(\text{Even}\), which holds on a randomly chosen finite structure with a probability approaching one as the size of the structure increases. However, Hella (1996) has shown that on unordered finite models, PTIME is not the extension of fixed-point logic by finitely many generalized quantifiers.

Recently there has been some interest in studying the computational complexity of generalized quantifiers in natural language. For example, Mostowski and Wojtyniak (2004) have observed that some natural language sentences, like (4), when assuming their branching interpretation, are mighty.

(4) Some relative of each villager and some relative of each townsman hate each other.

Sevenster (2006) has extended this results and proved that proportional branching quantifiers, like those possibly occurring in sentence (5), are also NP-complete.

(5) Most villagers and most townsmen hate each other.

We will overview these results in Chapter 3.

In the thesis we pursue the subject of the computational complexity of natural language quantifier constructions further. We prove some new results and study the role of descriptive computational complexity in natural language semantics.