Computational Complexity in Natural Language

Ian Pratt-Hartmann[1]

School of Computer Science,
University of Manchester,
Manchester M13 9PL, UK.
ipratt@cs.man.ac.uk

We have become so used to viewing natural language in computational terms that we need occasionally to remind ourselves of the methodological commitment this view entails. That commitment is this: we assume that to understand linguistic tasks—tasks such as recognizing sentences, determining their structure, extracting their meaning, and manipulating the information they contain—is to discover the algorithms required to perform those tasks, and to investigate their computational properties. To be sure, the physical realization of the corresponding processes in humans is a legitimate study too, but one from which the computational investigation of language may be pursued in Splendid Isolation. Complexity Theory is the mathematical study of the resources—both in time and space—required to perform computational tasks. What bounds can we place—from above or below—on the number of steps taken to compute such-and-such a function, or a function belonging to such-and-such a class? What bounds can we place on the amount of memory required? It is not surprising, therefore, that in the study of natural language, complexity-theoretic issues abound.

Since *any* computational task can be the object of complexity-theoretic investigation, it would be hopeless even to attempt a complete survey of Complexity Theory in the study of natural language. We focus therefore on a selection of topics in natural language where there has been a particular accumulation of complexity-theoretic results. Section 2 discusses parsing and recognition; Section 3 discusses the computation of logical form; and Section 4 discusses the problem of determining logical relationships between sentences in natural language. But we begin with a brief review of the Complexity Theory itself.

A draft chapter for the Blackwell *Computational Linguistics and Natural Language Processing Handbook*, edited by Alex Clark, Chris Fox and Shalom Lappin. This draft formatted on 2nd August 2009.

1 A brief review of Complexity Theory

Any account of Complexity Theory rests on some model of computation. The most widely-used such model is the multi-tape Turing machine; and that is the model we use here. Throughout this chapter, we employ standard notation for strings: if Σ is an alphabet (a finite, non-empty set of symbols), Σ^* denotes the set of *strings* (finite sequences of elements) over Σ. The length of any string σ is denoted $|\sigma|$; the empty (zero-length) string is denoted ϵ; and the concatenation of strings σ and τ is denoted $\sigma\tau$. We follow standard practice in ignoring the difference between elements of Σ and the corresponding 1-element strings.

1.1 Turing machines and models of computation

Informally, a *multi-tape Turing machine* comprises a finite number of *tapes*, a finite set of *states*, and an *instruction table.* The tapes may be thought of as the machine's memory, the states as the line numbers of its program, and the instruction table as the instructions of that program. The tapes are numbered consecutively from 1 to (say) $K \geq 2$; Tape 1 is referred to as the *input tape* and Tape K as the *output tape*; all other tapes are *work-tapes* (Fig. 1). Each tape consists of a one-way infinite sequence of squares (i.e. there is a left-most square, but no right-most square), and is scanned by its own *tape-head*, which is always located over one of these squares. Every square contains a unique symbol, which is either a member of a non-empty, finite set Σ, called the *alphabet* of the Turing machine, or one of the special symbols $\sqcup$ (read: 'blank') or $\triangleright$ (read: 'start').

The set of states, Q, is assumed to contain a pair of distinguished states: the *initial state* q_0 and the *halting state* q_1; otherwise, states have no internal structure. The instruction table of the Turing machine is a finite set T of quintuples

(1) $\langle p, \bar{s}, q, \bar{t}, \bar{d} \rangle$,

where p and q are states (i.e. elements of Q), $\bar{s} = (s_1, \ldots, s_K)$ and $\bar{t} = (t_1, \ldots, t_K)$ are K-tuples of symbols (i.e. elements of $\Sigma \cup \{\sqcup, \triangleright\}$), and $\bar{d} = (d_1, \ldots, d_K)$ is a K-tuple whose elements are the special tags `left`, `right` and `stay`. Informally, the Turing machine interprets the instruction (1) as follows:

(2) If the current state is p, and, for each k $(1 \leq k \leq K)$, the square currently being scanned on Tape k contains the symbol s_k, then set the new state to be q, and, for each k $(1 \leq k \leq K)$ do the following: write t_k on the square currently being scanned on Tape k, and place Tape k's head either one square left, or one square right, or in its current location, as directed by d_k.

We can make Tape 1 a read-only tape by insisting that it is never over-written; likewise, we can make Tape K a write-only tape by insisting that its head never moves to the left. The symbol $\triangleright$ is used to indicate the extreme left of a tape: we insist that, if any tape-head is over this symbol, it never receives an instruction to move left; moreover, $\triangleright$ is never written or over-written. The halting state q_1 indicates that the computation is over, and we insist that no instruction can be executed in this state. (It is easy to specify these conditions formally.) Technically speaking, a Turing machine is simply a tuple $M = \langle K, \Sigma, Q, q_0, q_1, T\rangle$ conforming to the above specifications.

Turing machines perform *computations*, which proceed in discrete time-steps. At each time-step, the machine is in a specific *configuration*, consisting of its current state q, the position of the tape head for each of the tapes, and the contents of each of the tapes. The initial configuration is as follows: the current state is q_0 (the initial state), with each tape-head positioned over the left-most square of the tape; Tape 0 has the symbol $\triangleright$ in the leftmost square, followed by a string $\sigma \in \Sigma^*$, called the *input* of the computation, and is otherwise filled with $\sqcup$; all other tapes have the symbol $\triangleright$ in the leftmost square, and are otherwise filled with $\sqcup$. At each time step, an instruction from T of the form (1) is executed as specified in (2), resulting in the next configuration. The computation halts when (and only when) no instruction in T can be executed. Note that, if the halting state q_1 is reached, the computation necessarily halts at that point. A *run* is a (finite or infinite) sequence of configurations obtained in this way; if the run is finite, so that the Turing machine halts, we call it a *terminating* run. Given a terminating run, the *output* of the computation is the string of Σ^* which, in the final configuration, is written on the output tape (strictly) between the $\triangleright$ and the first $\sqcup$. Notice that, in general, a Turing

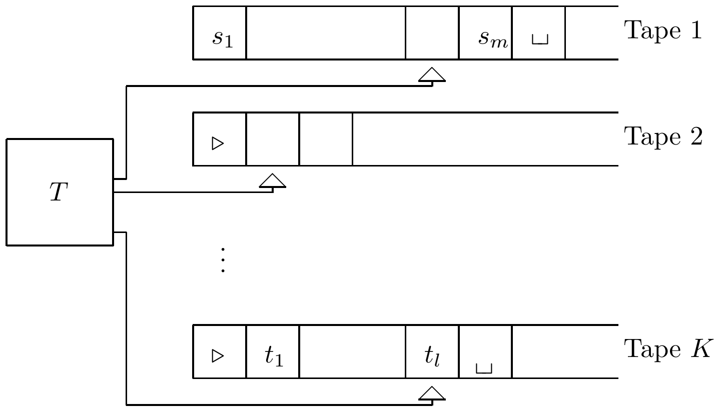

Figure 1. Architecture of a multi-tape Turing machine.

machine may be able to execute more than one instruction at any given time. In that case, we should think of the choice being made freely by the machine. We call a Turing machine *deterministic* just in case, for any state p and any K-tuple of symbols $\bar{s}$, T contains at most one instruction of the form (1) starting with the pair $\langle p, \bar{s}\rangle$ (i.e. the machine never has a choice as to which instruction to perform). A *non-deterministic Turing machine* is just another term for a Turing machine.

Definition 1 (Computable). *Let M be a deterministic Turing machine over alphabet Σ. For any string $\sigma \in \Sigma^*$, either M halts on input σ, or it does not. In the former case, M will output a definite string $\tau \in \Sigma^*$, and we can define the partial function $f_M : \Sigma^* \to \Sigma^*$ as follows.*

$$f_M(\sigma) = \begin{cases} \tau \textit{ if } M \textit{ halts on input } \sigma \\ \textit{undefined otherwise} \end{cases}$$

We say that M computes *the function f_M. A partial function $f : \Sigma^* \to \Sigma^*$ is* Turing computable *(or just:* computable*) if it is computed by some deterministic Turing machine.*

The instruction table of a Turing machine is fixed. Thus, a Turing machine is not a model of a computing machine in the sense we normally imagine, but rather of a computer program: there is only one thing it computes. On the other hand, since Turing machines are, formally, just tuples of finite objects, any Turing machine M can easily be coded as a string σ'_M over a suitable alphabet Σ', and that string can be input to another Turing machine, say M'. It can be shown that there exists a *universal Turing machine* U, which is able to simulate *any* Turing machine M over an alphabet Σ in the following sense: for any string, $\sigma \in \Sigma^*$, M has a non-terminating run on input σ if and only if U has a terminating run on input $\sigma'_M\sigma$; moreover, in case of termination, the output of M' is the same as the output of M. Any such Turing machine U *is* a model of a computing machine in the sense we normally imagine: it is able to execute an arbitrary 'program' σ'_M on arbitrary 'data' σ. Given such a coding scheme, consider the *halting function*, $H : (\Sigma')^* \to \{\top, \bot\}$ defined as

$$H(\sigma') = \begin{cases} \top \text{ if } \sigma' \text{ encodes a Turing machine } M \text{ that halts on input } \epsilon \\ \bot \text{ otherwise.} \end{cases}$$

This function is clearly well-defined, and indeed total. Perhaps the most fundamental fact in Computability Theory is due to Turing (1936–7):

Theorem 1 (Turing). *The halting function is not computable.*

Definition 1 applies to functions $f : \Sigma^* \to \Sigma^*$ for any alphabet Σ. However, this definition can be extended to functions with other countable domains and ranges, relative to some coding of the relevant inputs and outputs as strings over an alphabet. Consider for instance the familiar coding of natural numbers as bit strings (elements of $\{0, 1\}^*$). For $n \in \mathbb{N}$, denote by $\bar{n}$ the standard binary representation of n (without leading zeros). For $n \in \mathbb{N}$, denote by $\bar{n}$ the standard binary representation of n (without leading zeros); and for

$s \in \{0,1\}^*$, denote by $\#s$ the natural number represented by s. If $f : \mathbb{N} \to \mathbb{N}$ is a function, we consider f computable if the function $g : \{0,1\}^* \to \{0,1\}^*$ defined by

$$g(s) = \overline{(f(\#s))}$$

is computable in the sense of Definition 1. Computability of functions with other domains and ranges—e.g. rational numbers, lists, graphs *etc.*—is understood similarly. Technically, this extended notion of computability is relative to the coding scheme employed. In practice, however, all reasonable coding schemes usually yield the same computability (and complexity) results; if so, it is legitimate to speak of such functions as being computable or non-computable, leaving the operative coding scheme implicit.

The architecture of Turing machines given above is, in all essential details, that set out in Turing (1936–7). We have followed more recent practice in distinguishing input-, output- and work-tapes (Turing's machined had a single tape) to make it a little easier to talk about space-bounded computations. But this makes no difference to any of the results reported here. The thesis that Turing computability captures our pre-theoretic notion of computability is generally referred to as the *Church-Turing thesis*. It is important to appreciate that this thesis does not rest on the existence of universal Turing machines, or indeed on any purely mathematical fact. Methodologically, the apparatus introduced above is an exercise in conceptual analysis: the proposed replacement of an informally understood notion with a rigorous definition. Historically, several competing analyses of computability were proposed at more or less the same time, most notably Gödel's notion of *recursive function* and Church's λ-*calculus*. All three notions in effect coincide, however; so there is general consensus about the formal model presented here. For an accessible modern treatment, see Papadimitriou (1994, Chapter 2).

The fundamental goal of Complexity Theory is to analyse the computational resources, in either time or space, required to perform computational tasks. The first step is to measure the computational resources required by particular algorithms.

Definition 2. *Let M be a Turing machine with alphabet Σ, and let $g : \mathbb{N} \to \mathbb{N}$. We say M* runs in time *g if, for all but finitely many strings $\sigma \in \Sigma^*$, any run of M on input σ halts within at most $g(|\sigma|)$ steps. Similarly, M* runs in space *g if, for all but finitely many strings $\sigma \in \Sigma^*$, any run of M on input σ uses at most $g(|\sigma|)$ squares on any of its work-tapes.*

Allowing M to break the bound g in finitely many cases avoids problems caused by zero-length inputs and other trivial anomalies. Notice also the asymmetry in the definitions of time- and space-complexity: because measures of space complexity include only the *work-tapes* (and so exclude the input- and output-tapes), they can be *sub-linear*. For time-complexity, sub-linear bounds make little sense, because they do not give the machine the opportunity to read its input.

Unfortunately, Definition 2 is too fragile to provide a meaningful measure of algorithmic complexity. Suppose M is a deterministic Turing machine computing some function in time g, and let c be a positive number. Provided g is moderately fast-growing (say, faster than linear growth), it is routine to construct another deterministic Turing machine M'—perhaps with more tapes or more states or a larger alphabet—that computes the same function in time $cg(n)$. That is: we can always speed up M by a linear factor! Since M and M' do not represent interestingly different algorithms, the statement that a Turing machine runs in time—say—$3n^2+n+4$ as opposed to $14n^2+87n+11$ is, from an algorithmic point of view, not significant. Similar remarks also apply to space bounds.

Definition 3. *Let M be a Turing machine, and G a set of functions from $\mathbb{N}$ to $\mathbb{N}$. We say that M* runs in time G *if, for some $g \in G$, M runs in time g. Similarly, we say that M* runs in space G *if, for some $g \in G$, M runs in space g.*

In particular, the following classes of functions suggest themselves.

Definition 4 (O-notation). *Let $g : \mathbb{N}^k \to \mathbb{N}$ be a function. Denote by $O(g)$ the set of functions*

$$\begin{aligned} O(g) = \{g' : \mathbb{N}^k \to \mathbb{N} \mid \ & \textit{there exist } c \in \mathbb{N},\ n'_1, \ldots, n'_k \in \mathbb{N} \textit{ s.t.} \\ & \textit{for all } n_1 > n'_1 \ldots \textit{for all } n_k > n'_k,\ g'(n_1, \ldots, n_k) \leq cg(n_1, \ldots, n_k)\}. \end{aligned}$$

Informally, $O(g)$ is the class of functions which are eventually dominated by some positive multiple of g. Combining Definitions 3 and 4, it makes sense to say, for example, that a given Turing machine runs in time (or space) $O(n^2)$, or $O(n^3)$, or $O(2^n)$. And this sort of complexity-measure, it turns out, is *robust* under the expansions of computational resources considered above. For example, it can be shown that, for any $k > 0$, there is a function that can be computed by a deterministic Turing machine running in time $O(n^{k+1})$ which cannot be computed by any deterministic Turing machine running in time $O(n^k)$; and similarly for space-bounds. (The precise statement of these theorems, known as *separation theorems*, is somewhat intricate; see Kozen (2006, Lecture 3) or Papadimitriou (1994, pp.143 ff.). O-notation has the further advantage of permitting a useful degree of informality when analysing the complexity of an algorithm, since a pseudo-code description of that algorithm, of the sort standardly found in computing texts, often suffices to show that that it will run in time or space $O(g)$ (for some function g) without our having first to compile that description into a Turing machine. Finally, a word of caution. Knowing that a Turing machine (or algorithm) has time-complexity $O(g)$ at best imposes a bound on how rapidly the cost of computation grows with the size of the input. That is, the complexity-measures in question are *asymptotic*.

In many cases, algorithms with sub-optimal asymptotic complexity-measures perform best in practice.

1.2 Decision problems

So far, we have discussed complexity-measures for particular algorithms, understood as deterministic Turing machines. We now develop this idea in two crucial—though logically quite separate—ways.

The first development extends Definition 1 to *non-deterministic* computation. To do this, we first restrict attention to functions whose range contains just two elements—we conventionally employ $\top$ and $\bot$—representing 'YES' and 'NO', respectively. A function $f : A \to \{\top, \bot\}$, where A is a countable set, is called a *decision problem*, or simply a *problem*. While decision problems may initially seem of limited practical interest, they play a central role in Complexity Theory. Moreover, the restriction to decision problems is less severe than might at first appear: the complexity of many functions can often be usefully characterized in terms of the complexity of closely related decision problems.

Now, any decision problem $f : A \to \{\top, \bot\}$ can alternatively be regarded as a *subset* of A—namely, the subset $\{a \in A \mid f(a) = \top\}$. In particular, if $A = \Sigma^*$ for some alphabet Σ (or if the encoding of A in Σ^* is obvious), a decision problem defined on A is, in effect, a set of strings over Σ, or, in the parlance of Formal Language Theory, a *language* over Σ. Conversely, of course, any language $L \subseteq \Sigma^*$ may be regarded as a decision problem $f : \Sigma^* \to \{\top, \bot\}$ given by:

$$f(\sigma) = \begin{cases} \top & \text{if } \sigma \in L \\ \bot & \text{otherwise.} \end{cases}$$

The observation that decision problems and languages are essentially the same thing prompts the following definition.

Definition 5. *Let M be a Turing machine over the alphabet Σ, and suppose without loss of generality that Σ contains the symbol $\top$. We say that M* accepts *a string $\sigma \in \Sigma^*$ if there exists a terminating run of M with input σ and output $\top$. The language $L \subseteq \Sigma^*$* recognized by *M, denoted $L(M)$, is the set of strings accepted by M.*

It is important to bear in mind that, in Definition 5, M can be *non-deterministic*. That is: $L(M)$ is the set of inputs for which M *may* yield the output $\top$. (It is sometimes convenient to imagine a benign helper guiding M to make the 'right' choice of instructions required to accept a string $\sigma \in L$.) Equally important is that, if $\sigma \notin L$, there is no requirement for M to produce any particular output (as long as it is not $\top$, of course), or indeed to halt at all.

The case where M halts on every input is of particular interest, however:

Definition 6 (Decidable). *Let L be a language. We call L* decidable *if it is recognized by a Turing machine that halts on every input.*

It is routine to show that any decidable language is in fact recognized by a *deterministic* Turing machine that halts on every input. Furthermore, that machine can easily be modified so as always to produce one of the two outputs $\top$, $\bot$. Thus, a decision problem $f : \Sigma^* \rightarrow \{\top, \bot\}$ is a computable function, in the sense of Definition 1, just in case the corresponding language $L = \{\sigma | f(\sigma) = \top\}$ is decidable, in the sense of Definition 6. Henceforth, then, we shall identify decision problems and languages, employing whichever term is most appropriate in context.

We may think of Definition 5 as a generalization of Definition 1 to the case of *non-deterministic* computation. The significance of this generalization is that, while deterministic and non-deterministic Turing machines recognize the same class of languages, they may not in general do so within the same computational bounds, a possibility which plays a central role in Complexity Theory.

We can generalize the above observations on linear speedup to the case of non-deterministic computation for decision problems. We give a reasonably precise version here:

Theorem 2. *Let L be a language over some alphabet, let $g : \mathbb{N} \rightarrow \mathbb{N}$ and $h : \mathbb{N} \rightarrow \mathbb{N}$ be functions, let $c \geq 1$, and suppose $g(n) \geq n+1$, and $h(n) \geq \log n$. If L is recognized by some Turing machine running in time $cg(n)$, then it is recognized by some Turing machine running in time $g(n)$. If L is recognized by some Turing machine running in space $ch(n)$, then it is recognized by some Turing machine running in space $h(n)$. The previous statements continue to hold when "Turing machine" is replaced throughout by "deterministic Turing machine".*

Now for the second development in our analysis of complexity. So far, we have provided measures of the time- and space-requirements of particular *Turing machines* (or, by extension, and using O-notation, of particular *algorithms*). But what primarily interests us in Complexity Theory are the time- and space-requirements of a *maximally efficient* Turing machine for computing a particular *function* or, more specifically, solving a particular *decision problem.* Recalling the equivalence between decision problems and languages discussed above, we define:

Definition 7. *Let L be a language over some alphabet, and let G be a set of functions from $\mathbb{N}$ to $\mathbb{N}$. We say that L* is in *TIME*(G) *(or SPACE*(G)*) if there exists a deterministic Turing machine M recognizing L, such that M runs in time (respectively, space) G.*

Classes of languages of the form TIME(G) or SPACE(G) are referred to as (deterministic) complexity classes. To avoid notational clutter, if g is a function from $\mathbb{N}$ to $\mathbb{N}$, we write TIME(g) instead of TIME$(\{g\})$; and similarly for other complexity classes.

So far, we have encountered classes of functions of the form $O(g)$ for various g. When analysing the complexity of *languages* (rather than of specific algorithms), however, larger classes of functions are typically more useful.

Definition 8. *Let P, E, and E_k (for $k > 1$) be the sets of functions from $\mathbb{N}$ to $\mathbb{N}$ defined as follows:*

$$\begin{aligned} P &= \{n^c \mid c > 0\} \\ E &= \{2^{n^c} \mid c > 0\} \\ E_2 &= \{2^{2^{n^c}} \mid c > 0\} \\ E_k &= \{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\Big\}^{n^c} k \textit{ times}} \mid c > 0\} \end{aligned}$$

A function $g : \mathbb{N} \to \mathbb{N}$ which is in E_k for some k is said to be elementary.

Non-elementary functions grow rapidly. However, it is easy to define a computable function which is non-elementary:

$$f(n) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\Big\}n \text{ times}} .$$

Combining Definitions 7 and 8, we obtain complexity classes which are often known under the following, more pronounceable names:

$$\begin{array}{rr} & \text{LOGSPACE} = \text{SPACE}(\log n) \\ \text{PTIME} = \text{TIME}(P) & \text{PSPACE} = \text{SPACE}(P) \\ \text{EXPTIME} = \text{TIME}(E) & \text{EXPSPACE} = \text{SPACE}(E) \\ k\text{-EXPTIME} = \text{TIME}(E_k) & k\text{-EXPSPACE} = \text{SPACE}(E_k). \end{array}$$

Thus, PTIME is the class of languages recognizable by a deterministic Turing machine in polynomial time, EXPSPACE, the class of languages recognizable by a deterministic Turing machine in exponential space, and so on. In some texts, LOGSPACE is referred to as L, PTIME as P, and EXPTIME as EXP. Notice, incidentally, that there is no point in defining, say, $G = \{\log(n^c) \mid c > 0\}$ and then setting LOGSPACE = SPACE(G), since, by Theorem 2, linear factors may be ignored. Finally, if L is not recognizable by any Turing machine running in time bounded by an elementary function, then L is said to have *non-elementary complexity*. We shall encounter examples of decidable, but non-elementary, problems below.

Definition 7 may be adapted directly to deal with non-deterministic computation.

Definition 9. *Let L be a language over some alphabet, and let G be a set of functions from $\mathbb{N}$ to $\mathbb{N}$. We say that L* is in *NTIME(G) (or NSPACE(G)) if there exists a Turing machine M recognizing L, such that M runs in time (respectively, space) G.*

Classes of languages of the form NTIME(G) or NSPACE(G) are referred to as (non-deterministic) complexity classes.

Combining Definitions 8 and 9, we obtain complexity classes which are often known under the following, more pronounceable names:

$$
\begin{array}{rr}
 & \text{NLOGSPACE} = \text{NSPACE}(\log n) \\
\text{NPTIME} = \text{NTIME}(P) & \text{NPSPACE} = \text{NSPACE}(P) \\
\text{NEXPTIME} = \text{NTIME}(E) & \text{NEXPSPACE} = \text{NSPACE}(E) \\
\text{N}k\text{-EXPTIME} = \text{NTIME}(E_k) & \text{N}k\text{-EXPSPACE} = \text{NSPACE}(E_k).
\end{array}
\tag{3}
$$

In some texts, NLOGSPACE is referred to as NL, NPTIME as NP, and NEXPTIME as NEXP.

Notice the asymmetry involved in the notion of non-deterministic computation: M recognizes $L \subseteq \Sigma^*$ just in case, for each string $\sigma \in \Sigma^*$, $\sigma \in L$ if and only if *there exists* a successfully terminating run of M (i.e. a terminating run with output $\top$) on input σ—that is to say, $\sigma \in \Sigma^* \setminus L$ if and only if *all* runs of M on input σ fail to halt successfully. This asymmetry prompts us to define the *complement* classes as follows.

Definition 10. *If $\mathcal{C}$ is a class of languages, then* Co-$\mathcal{C}$ *is the class of languages L such that $\Sigma^* \setminus L$ is in $\mathcal{C}$, where Σ is the alphabet of L.*

It is easy to see that, for any interesting class of functions G, TIME(G)= Co-TIME(G) and SPACE(G)= Co-SPACE(G). For this reason, we never speak of Co-PTIME, Co-PSPACE, *etc.* The situation with non-deterministic complexity classes is different, however. It is not known whether NPTIME= Co-NPTIME; and similarly for many other classes of the form Co-NTIME(G). Indeed, such complexity classes are regularly encountered. In particular, putting together Definition 10, and the NTIME-classes listed in (3), we obtain the complexity classes Co-NPTIME, Co-NEXPTIME and Co-Nk-EXPTIME. (And similarly for the corresponding space-complexity classes; but see Theorem 4.)

1.3 Relations between complexity classes

It is obvious from the above definitions that any language in TIME(G) (or SPACE(G)) is non-deterministically recognizable within the same bounds. Formally,

$$\text{TIME}(G) \subseteq \text{NTIME}(G) \qquad \text{SPACE}(G) \subseteq \text{NSPACE}(G).$$

A little less obviously, we see that

$$\text{NPTIME} \subseteq \text{EXPTIME} \qquad \text{NEXPTIME} \subseteq \text{2-EXPTIME} \qquad \cdots$$

Consider the first of these inclusions. If M non-deterministically recognizes L, and p is a polynomial such that M is guaranteed to halt within time $p(n)$ on input of size n, the number of possible runs of M on inputs of this size is easily seen to be bounded by $2^{q(n)}$ for some polynomial q. But then a deterministic Turing machine M', simulating M, can check all of these runs in exponential time, outputting $\top$ if any one of them halts successfully. Hence, NPTIME $\subseteq$ EXPTIME. The inclusion NEXPTIME $\subseteq$ 2-EXPTIME follows analogously; and so on up the complexity hierarchy. In fact, similar arguments establish the following more elaborate system of inclusions.

$$\begin{aligned}&\text{PTIME} \subseteq \text{NPTIME} \subseteq \text{PSPACE} \subseteq\\ &\quad\text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE} \subseteq\\ &\qquad\text{2-EXPTIME} \subseteq \text{2-EXPTIME} \cdots\end{aligned} \tag{4}$$

The following result establishes that, for classes of sufficiently ‘large’ functions, non-determinism makes no difference to space-complexity (Savitch, 1970).

Theorem 3 (Savitch). *If $g(n) \geq \log n$, then*

$$\text{NSPACE}(g(n)) \subseteq \text{SPACE}((g(n))^2).$$

In some statements of this theorem, certain technical conditions are imposed on g; but see, e.g. Kozen (2006, pp. 15–16). Since the classes of functions P, E, E_2, etc. are closed under squaring, we have NPSPACE=PSPACE, NEXPSPACE=EXPSPACE, and so on. As an instant corollary, since these deterministic classes are equal to their complements, we have NPSPACE = Co-NPSPACE, NEXPSPACE=Co-NEXPSPACE, and so on.

Care is required when applying the reasoning of the previous paragraph. Setting $g(n) = \log n$, Theorem 3 tells us that NLOGSPACE $\subseteq$ SPACE$((\log n)^2)$; however, this is not sufficient to imply that NLOGSPACE $\subseteq$ LOGPSPACE. Nevertheless, the following result establishes that equivalence under complementation continues to hold even in this case (Immerman, 1988).

Theorem 4 (Immerman-Szelepcsényi). *If $g(n) \geq \log n$, then*

$$\text{NSPACE}(g(n)) = \text{Co-NSPACE}(g(n)).$$

In some statements of this theorem, certain technical conditions are imposed on g; but again, see Kozen (2006, pp. 22–24). As a special case, we have NSPACE(n) = Co-NSPACE(n), which settled a long-standing conjecture in Formal Language Theory (see Section 2.3 below). As an instant corollary of Theorem 4, NLOGSPACE=Co-NLOGSPACE.

Adding these ‘small’ complexity classes to the inclusions (4), we obtain

$$\begin{aligned}&\text{LOGSPACE} \subseteq \text{NLOGSPACE} \subseteq \text{PTIME} \subseteq \text{NPTIME} \subseteq\\ &\quad\text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq\\ &\qquad\text{EXPSPACE} \subseteq \text{2-EXPTIME} \subseteq \text{2-NEXPTIME} \cdots\end{aligned} \tag{5}$$

1.4 Lower bounds

Notwithstanding the above caveats on the interpretation of asymptotic complexity-measures, saying that that a language is in a complexity class $\mathcal{C}$ places some kind of *upper bound* on the resources required to recognize it. But what of *lower bounds*? What if we want to say that a language *cannot* be recognized within certain time- or space-bounds? For the complexity classes introduced above, useful lower-bound characterizations are indeed possible.

The basic idea is that of a *reduction* of one language (or decision problem) to another. Let L_1 and L_2 be languages, perhaps over different alphabets Σ_1 and Σ_2. Suppose that there exists a function $g : \Sigma_1^* \rightarrow \Sigma_2^*$ such that, for any string $\sigma \in \Sigma_1^*$, $\sigma \in L_1$ if and only if $g(\sigma) \in L_2$. We may think of g as a means of 'translating' L_1 into L_2: in particular, any Turing machine recognizing L_2 can be modified to recognize L_1 by simply prepending the translation g. If the cost of this translation is small, then we may regard L_2 as being 'at least as hard to recognize as' L_1.

Definition 11 (Reduction). *Let Σ_1 and Σ_2 be alphabets, and let L_i be a language over Σ_i $(i = 1, 2)$. A* reduction of L_1 to L_2 *is a function $g : \Sigma_1^* \rightarrow \Sigma_2^*$, such that g can be computed by a Turing machine in space $O(\log n)$, and for all $\sigma \in \Sigma_1^*$, $\sigma \in L_1$ if and only if $g(\sigma) \in L_2$; in that case, we say that L_1 is* reducible to L_2. *If, instead, g can merely be computed in time $O(n^k)$ for some k, we call it a* polynomial reduction*, and we say that L_1 is* polynomially reducible to L_2.

Let $\mathcal{C}$ be any of the complexity classes mentioned in (5), or the complement of any of these classes. It is easy to see that, if L_2 is in $\mathcal{C}$, and L_1 is reducible to L_2, then L_1 in $\mathcal{C}$. We say that $\mathcal{C}$ is 'closed under reductions'. If $\mathcal{C}$ is any of the complexity classes mentioned in (4), then $\mathcal{C}$ is, similarly, 'closed under polynomial reductions'.

Theorem 5. *The relation of reducibility is transitive: if L_1 is reducible to L_2, and L_2 to L_3, then L_1 is reducible to L_3.*

We remark that Theorem 5 is not obvious (though its analogue in the case of *polynomial* reducibility is): see, e.g. Papadimitriou (1994, p. 164).

Now we can give our characterization of lower complexity bounds.

Definition 12 (Hardness and completeness). *Let $\mathcal{C}$ be a complexity class. A language L is said to be* hard for $\mathcal{C}$*, or* $\mathcal{C}$-hard*, if any language in $\mathcal{C}$ is reducible to L; L is said to be* complete for $\mathcal{C}$*, or* $\mathcal{C}$-complete*, if L is $\mathcal{C}$-hard and also in $\mathcal{C}$. Additionally, L is said to be* $\mathcal{C}$-hard under polynomial reduction *if any decision problem in $\mathcal{C}$ is polynomially reducible to L; similarly for* $\mathcal{C}$-completeness under polynomial reduction

It follows from Theorem 5 that, if L_1 is $\mathcal{C}$-hard for some complexity class $\mathcal{C}$, and L_1 is reducible to L_2, then L_2 is $\mathcal{C}$-hard. Similarly, *mutatis mutandis*, for hardness under polynomial reductions. Notice that the notion of LOGSPACE-completeness is uninteresting: any problem in LOGSPACE is by definition LOGSPACE-complete. Under polynomial reductions, the notion of PTIME-completeness is similarly uninteresting. Definition 12 reflects the fact that reducibility in logarithmic space is taken to be the default in Complexity Theory. However, for most higher complexity classes, it is generally easier and just as informative to work with reducibility in polynomial time; and this is what is often done in practice. Hardness results, in the sense of Definition 12, are sometimes referred to, for obvious reasons, as 'lower complexity bounds'. However, it is important not to be misled by this terminology: for example, it is easy to show that there are PTIME-hard problems in TIME(n); but TIME(n) is properly contained in PTIME!

Many natural problems (it is easier here to speak of problems rather than languages) can be shown to to be complete for the complexity classes introduced above. Here are three very well-known examples. In the context of propositional logic, a *literal* is a proposition letter or a negated proposition letter; proposition letters are said to be *positive* literals, their negations *negative* literals. A *clause* is a disjunction of literals; a clause is said to be *Horn* if it contains at most one positive literal. Theorems 6–9 are among the most fundamental in Complexity Theory. For an accessible treatment, see, e.g. Papadimitriou (1994, p. 171, p. 176, p. 398, respectively). Theorem 6 is due to Cook (1971).

Theorem 6 (Cook). *The problem of determining whether a given set of clauses is satisfiable is NPTIME-complete.*

Theorem 7. *The problem of determining whether a given set of Horn clauses is satisfiable is PTIME-complete.*

Theorem 8. *The problem of determining the satisfiability of a given set of clauses, all of which contain most two literals, is NLOGSPACE-complete.*

Theorem 8 is very closely related to following graph-theoretical problem. Given a finite directed graph, one node in that graph is said to be *reachable* from another if there is a finite sequence of directed edges in that graph leading from the first node to the second.

Theorem 9. *The problem of determining whether, in a given directed graph, one node is reachable from another, is NLOGSPACE-complete.*

Note that, in each case, we assume that inputs (clauses, graphs, ...) are coded in some standard way as strings over some alphabet. All reasonable coding schemes yield the same complexity results.

Such completeness results are often less surprising than they at first appear. For example, Theorem 6 is established by showing that, given a non-deterministic Turing machine M that runs in polynomial time, the conditions for a sequence of configurations of M to be a run of M with input σ can be encoded, in a natural way, as a set of clauses whose size is bounded by a polynomial function of the length of σ. And once one language L is shown to be hard for a complexity class, other languages can be shown to be hard for that class by showing that L is reducible to them.

2 Parsing and recognition

As already mentioned, in the context of Formal Language Theory, a *language* is a set of strings over some alphabet Σ. Some languages are specified by *grammars*, which are themselves finite objects whose semantics is defined by a *grammar framework*. Familiar grammar frameworks are: context-sensitive grammars, definite clause grammars, tree-adjoining grammars, context-free grammars and non-deterministic finite-state automata. Within a given grammar framework $\mathcal{F}$, any grammar G *recognizes* a unique language $L(G)$, namely, the set of strings *accepted* by G. Thus, the apparatus of the multi-tape Turing machine also constitutes a grammar framework in this sense. Each grammar in that framework—that is, each specific Turing machine M over signature Σ—recognizes the language $L(M)$ comprising the set of strings over Σ accepted by M, in the sense of Definition 5.

If $\mathcal{F}$ is a grammar framework, we understand the *universal recognition problem* for $\mathcal{F}$ to be the following problem: given a grammar G in $\mathcal{F}$ and a string σ over the alphabet of G, determine whether $\sigma \in L(G)$. This problem is to be distinguished from the *fixed-language recognition problem* for any G in $\mathcal{F}$: given a string σ over the alphabet of G, determine whether $\sigma \in L(G)$. The complexity of the universal recognition problem for a framework $\mathcal{F}$ is in general higher than that of the fixed-language recognition problem for any grammar in $\mathcal{F}$.

In this section, we survey the complexity of the universal recognition problem and the fixed language recognition problem for various grammar frameworks. For the framework of Turing machines, we already know the answer: it is (essentially) a restatement of Theorem 1 that the universal recognition problem for Turing Machines is undecidable; and it is an immediate consequence of the existence of a universal Turing Machine that there exist Turing machines whose fixed language recognition problem is undecidable. For less expressive grammar frameworks, however, there is much more to be said, as we shall see.

2.1 Regular languages

Let us begin with one of the least expressive of the commonly encountered grammar frameworks. A *non-deterministic finite-state automaton* (NFSA) is a tuple $\mathcal{A} = \langle \Sigma, Q, q_0, q_1, T \rangle$, where Σ is an alphabet, Q a set (the set of *states* of $\mathcal{A}$), q_0 and q_1 distinct elements of Q (the *initial state* and the *accepting state* respectively), and T a finite set of triples $\langle p, s, q \rangle$, (the *transitions* of $\mathcal{A}$), where $p, q \in Q$ and $s \in \Sigma$. Informally, the transition $\langle p, s, q \rangle$ has the interpretation

> If the current state is p, and the next symbol to be read is s, then set the new state to be q.

An NFSA $\mathcal{A}$ is said to *accept* the string $\sigma = s_1, \ldots, s_n$ if, starting in the state q_0, and reading the symbols $s_1, \ldots, s_n$ successively, there is a sequence of transitions in T leading to the state q_1. NFSAs may be pictured as labelled graphs in the obvious way: the nodes are labelled by elements of Q, and the edges by elements of Σ. A string is accepted if it is possible to step through the graph from the initial state to the final state in such a way that the string is exactly consumed.

It is a standard result of Formal Language Theory that the class of languages accepted by NFSAs coincides with the class of regular languages. A *regular expression* over an alphabet Σ is defined recursively to be any expression of the forms $\emptyset$, ϵ, s, $e_1 \cup e_2$, $e_1 e_2$ or e^*, where $s \in \Sigma$ and e, e_1 and e_2 are regular expressions. Any regular expression e recognizes a language $L(e)$ over Σ, defined (with harmless abuse of notation) as follows:

$$\begin{array}{l} L(\emptyset) = \emptyset \qquad L(\epsilon) = \{\epsilon\} \qquad L(s) = \{s\} \text{ for } s \in \Sigma \\ L(e_1 \cup e_2) = L(e_1) \cup L(e_2) \\ L(e_1 e_2) = \{\sigma\tau \mid \sigma \in L(e_1) \text{ and } \tau \in L(e_2)\} \\ L(e^*) = \{\sigma_1 \ldots \sigma_k \mid k \geq 0 \text{ and } \sigma_i \in L(e) \text{ for all } i \ (1 \leq i \leq k)\}. \end{array}$$

A *regular language* is any language $L(e)$, where e is a regular expression.

Deciding whether a given NFSA accepts a given string is easily reducible to the problem of reachability in directed graphs, and *vice versa*. By Theorem 9, therefore, we have

Theorem 10. *The universal recognition problem for NFSAs is NLOGSPACE-complete.*

What about the fixed-language recognition problem? An NFSA can be thought of as a Turing machine with a finite memory—that is, a Turing machine which never uses more than a constant amount of space on any of its work-tapes. With a little care, this equivalence can be shown to be exact: a language is regular if and only if it can be recognized by a Turing machine with fixed space bound. Hence:

Theorem 11. *For any NFSA $\mathcal{A}$, $L(\mathcal{A})$ is in SPACE(c) for come constant c.*

Thus, the universal recognition problem for NFSAs has higher complexity than the recognition problem for any specific regular language. A subtly different illustration of this phenomenon is provided by the grammar framework of extended regular expressions. An *extended regular expression* over an alphabet Σ is defined exactly as for regular expressions, except that we have a complementation operator $\bar{e}$, with semantics given by

$$L(\bar{e}) = \Sigma^* \setminus L(e).$$

A well-known theorem of Formal Language Theory states that the class of regular languages is closed under complementation, and hence is equal to the class of languages recognized by extended regular expressions. Thus, the grammar frameworks of NFSAs and extended regular expressions are equal in expressive power. However, the corresponding universal recognition problems may have different complexity. Stockmeyer & Meyer (1973, p. 3) show:

Theorem 12. *The universal recognition problem for extended regular expressions is in PTIME.*

Theorem 12 does not easily follow from Theorem 10: extended regular expressions constitute a more compact way of specifying regular languages than do NFSAs. Of course, when it comes to the fixed-language recognition problem for languages defined by extended regular expressions, this must be the same as for NFSAs, because they are the same languages. For a useful list of complexity-theoretic results regarding regular languages, see Yu (1997, pp. 96 ff.).

2.2 Context-free languages

Probably the most familiar and useful grammar framework in linguistics is that of context-free grammars. Formally, A *context-free grammar* (CFG) is a quadruple $G = \langle N, \Sigma, S, P \rangle$, where N is a set of *non-terminals* (typically, category labels such as S, NP, VP etc.), Σ an alphabet, S a distinguished *start-symbol* in N (for example, the category S), and P a list of *productions* for re-writing non-terminals (such as, S $\rightarrow$ NP VP, NP $\rightarrow$ Det N, etc.). Elements of Σ are usually referred to as *terminals* in this context. A CFG accepts the string of terminals σ if some sequence of productions can be found which rewrites the start–symbol S to σ. A language recognized by a CFG is called a *context-free language*. For example, the language $\{\mathrm{a}^n \mathrm{b}^n \mid n \geq 0\}$ is context-free, but not regular. (For a detailed discussion, see Chapter 1, Section 6.) HARD REF

A number of well-known algorithms exist to determine whether, given a CFG G and a string σ, G accepts σ. Perhaps the best-known is the *CYK* algorithm, named after its simultaneous inventors, Cocke, Younger and Kasami (see, e.g. Younger (1967)). Under reasonable assumptions about what qualifies as a constant-time operation, this algorithm runs in time $O(mn^3)$, where m is the number of productions in G, and n is the length of σ; however it requires that the given grammar G be in Chomsky normal form. The slightly more sophisticated algorithm of Earley (1970) dispenses with this assumption. Thus, the universal recognition problem for context-free languages is in PTIME. Furthermore, it is easy to reduce this problem to the satisfiability problem for Horn-clauses in propositional logic, whence, by Theorem 7, it is also PTIME-hard (Jones & Laaser, 1977). Hence:

Theorem 13. *The universal recognition problem for CFGs is PTIME-complete.*

On the other hand, for the fixed-language recognition problem, we can again do a little better (Ritchie & Springsteel, 1972; Nepomnyashchii, 1975):

Theorem 14. *For any CFG G, $L(G)$ is in $SPACE((\log n)^2)$. Moreover, there exists a context-free language which is NLOGSPACE-hard.*

The proof in both cases is rather technical.

CFGs are not the only way of describing context-free languages: the framework of *Lambek grammars* (Lambek, 1958) provides an alternative. We content ourselves with an informal explanation here, referring the reader to, e.g. Carpenter (1997). The *Lambek calculus* (*with product*) is a logical system allowing the derivation of *sequents* involving *category expressions.* A category expression is either a *basic category* or a *derived category* of the forms X/Y, $Y\backslash X$ or $X \cdot Y$. Examples of basic categories are S and NP. Examples of derived categories are NP\S, NP · NP and (NP\S)/NP . Intuitively, a category expression X/Y describes a string which, when a string of category Y is placed to its *right*, will result in a string of category X; similarly, $Y\backslash X$ describes a string which, when a string of category Y is placed to its *left*, will result in a string of category X; and finally, $X \cdot Y$ describes a string which is the result of concatenating a string of category X and a string of category Y. Thus, an intransitive verb, and indeed any verb phrase, might be assigned category NP\S, while a transitive verb might be assigned category (NP\S)/NP.

A *sequent* in the Lambek calculus is an expression of the form

$$X_1 \cdots X_n \to X,$$

where X_1, ..., X_n and X are category expressions. Intuitively, such a sequent has the meaning: "The result of concatenating any strings of categories $X_1, \ldots, X_n$, in that order, is a string of category X. An example of a sequent is

(6) NP (NP\S)/NP NP → S,

which thus has the informal interpretation

(7) if σ_1, σ_2 and σ_3 are strings of categories NP , (NP\S)/NP and NP, respectively, then $\sigma_1\sigma_2\sigma_3$ is of category S.

We remark that, under the advertised interpretations of the relevant derived categories, (7) is a true statement. Formally, however, it is the rules of the Lambek calculus (rather than judgments such as (7)) that determine whether any given sequent is derivable. We do not give these rules here. As an example, however, Fig. 2 shows the derivation of Sequent (6). It can be shown that the

$$\dfrac{\dfrac{\mathrm{S} \to \mathrm{S} \qquad \mathrm{NP} \to \mathrm{NP}}{\mathrm{NP} \;\; (\mathrm{NP}\backslash\mathrm{S}) \to \mathrm{S}}\,(\backslash\mathrm{I}) \qquad \mathrm{NP} \to \mathrm{NP}}{\mathrm{NP} \;\; (\mathrm{NP}\backslash\mathrm{S})/\mathrm{NP} \;\; \mathrm{NP} \to \mathrm{S}}\,(/\mathrm{I})$$

Figure 2. A derivation in the Lambek calculus.

rules of the Lambek calculus are correct and complete for the interpretation given above (Pentus, 1994).

A *Lambek grammar* (*with product*) over a signature Σ is a finite list G of pairs of the form (s, C) where $s \in \Sigma$ and C is a category expression. We say that the grammar G *accepts* the string $\sigma = s_1, \ldots, s_n$ just in case there exist category expressions $C_1, \ldots, C_n$ such that: (i) $(s_i, C_i) \in G$ for each i $(1 \leq i \leq n)$, and (ii) the sequent $C_1 \cdots C_n \to S$ can be derived in the Lambek calculus. (Again, S is a distinguished start-symbol.) For example, if G contains the pairs

$$(\mathsf{John}, \mathrm{NP}), \qquad (\mathsf{Mary}, \mathrm{NP}), \qquad (\mathsf{loves}, (\mathrm{NP}\backslash\mathrm{S})/\mathrm{NP}),$$

then, since (6) is a valid sequent, G accepts the sentence John loves Mary. It is known (Pentus, 1993, 1997) that the class of languages recognized by Lambek grammars is exactly the class of context-free languages.

A crucial result concerning the Lambek calculus is the so-called *cut-elimination theorem* (Lambek, 1958), which allows us to show that the problem of determining the validity of a given sequent in the Lambek calculus is in NPTIME. More recently, Pentus (2006) has shown that the problem of determining the validity of a sequent in the Lambek calculus (with product) is NPTIME-complete. This immediately translates, in the present context, to the following result.

Theorem 15 (Pentus). *The universal recognition problem for Lambek grammars* (*with product*) *is NPTIME-complete.*

We remark in passing that the corresponding problem for the Lambek calculus without the operation $\cdot$ (i.e. just the operations $/$ and $\backslash$) is, at the time of writing, open. This restriction does not decrease the class of languages which can be recognized by such grammars: these are still exactly the context-free languages.

2.3 More expressive grammar frameworks

As the preceding discussion illustrates, the complexity of the universal recognition problem for a grammar framework cannot be read off in any simple way from its expressive power. Nevertheless, commonly encountered grammar frameworks with higher expressive power do tend, by and large, to exhibit higher recognition complexity. A well-known example is provided by the class of *tree-adjoining grammars* (TAGs). A more detailed explanation of TAGs can be found in Chapter 4, Section 7. But very roughly, a TAG is a finite set of 'local' trees which can be combined into larger trees to license sentences, much as CFGs combine productions (which can equally be thought of as local trees) into phrase-structures. The essentially new element in TAGs is the operation of *adjunction*, in which a local tree may be 'spliced' into an existing tree. A language recognized by a TAG is called a *tree-adjoining language*. HARD REF

The more elaborate apparatus of TAGs leads to an increase in recognition capacity: the languages $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n \mid n \geq 0\}$ and $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n\mathrm{d}^n \mid n \geq 0\}$ are tree-adjoining languages, but not context-free languages. It also leads to an increase in recognition complexity. Various parsing algorithms have been developed which show that the recognition problem for a TAG can be solved in time $O(n^6)$, where n is the length of the input string (Schabes, 1994). Interestingly, TAGs turn out to be expressively equivalent to several other natural grammar frameworks, including *head grammars*, *linear index grammars* and *combinatory categorial grammars* (Vijay-Shanker & Weir, 1994). These equivalences can be used to establish that all these grammar frameworks have universal recognition problems with comparable complexity.

More expressive still is the framework of *definite clause grammars* (DCGs). Again, we give only an informal explanation here. Like a CFG, a DCG consists of a set of productions over fixed sets of terminal and non-terminal symbols, together with a distinguished non-terminal S. The only difference is that the non-terminals now take *arguments* drawn from a *term-language* $\mathcal{T}$. The expressions of $\mathcal{T}$ are built up from a fixed vocabulary of individual constants, variables, and function-symbols. We assume that there is at least one individual constant in $\mathcal{T}$. Each non-terminal in a DCG is associated with a non-negative integer, called its *arity*, and, in any production, is supplied with a list of arguments according to that arity. A typical DCG production has the form

$$(8)\quad A(s_1, \ldots, s_n) \rightarrow B_1(t_{1,1}, \ldots, t_{1,\ell_1}) \ \cdots \ B_m(t_{m,1}, \ldots, t_{m,\ell_m}),$$

where A is a non-terminal with arity n, and the B_i are non-terminals with arity ℓ_i for all i ($1 \leq i \leq m$). (In general, the right-hand side is also allowed to contain terminals.) The distinguished non-terminal S is assumed to have arity 0. A *ground instance* of a production is the result of consistently substituting, for the variables in that production, terms which contain no variables. The notion of *acceptance* is then defined in the same way as for a CFG, by regarding each production as the set of its ground instances. (Of course, this set of productions may be infinite, and thus will not in general constitute an actual CFG.)

Fig. 3 shows a set of productions for a DCG G with non-terminals $\{\mathrm{S}, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}\}$ and terminals $\{\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d}, \mathrm{e}\}$. Each of the non-terminals has arity 1, except for S, the distinguished non-terminal. Fig. 4 shows a derivation

$$\mathrm{S} \rightarrow \mathrm{A}(x)\ \mathrm{B}(x)\ \mathrm{C}(x)\ \mathrm{D}(x)\ \mathrm{E}(x)$$

$$\begin{array}{ll}
\mathrm{A}(1) \rightarrow \mathrm{a} & \mathrm{A}(\mathrm{f}(x)) \rightarrow \mathrm{a}\ \mathrm{A}(x) \\
\mathrm{B}(1) \rightarrow \mathrm{b} & \mathrm{B}(\mathrm{f}(x)) \rightarrow \mathrm{b}\ \mathrm{B}(x) \\
\mathrm{C}(1) \rightarrow \mathrm{c} & \mathrm{C}(\mathrm{f}(x)) \rightarrow \mathrm{c}\ \mathrm{C}(x) \\
\mathrm{D}(1) \rightarrow \mathrm{d} & \mathrm{D}(\mathrm{f}(x)) \rightarrow \mathrm{d}\ \mathrm{D}(x) \\
\mathrm{E}(1) \rightarrow \mathrm{e} & \mathrm{E}(\mathrm{f}(x)) \rightarrow \mathrm{e}\ \mathrm{E}(x)
\end{array}$$

Figure 3. Productions of a DCG recognizing the language $\{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n\mathrm{d}^n\mathrm{e}^n \mid n \geq 0\}$.

of the string aabbccddee in G, where the variable x in the first production takes the value f(1). This variable in effect counts the number of times the rules for the non-terminals A, ..., E are invoked (with a value $f^{n-1}(1)$ encoding n invocations), and ensures that this number is the same in each case. Thus, $L(G) = \{\mathrm{a}^n\mathrm{b}^n\mathrm{c}^n\mathrm{d}^n\mathrm{e}^n \mid n \geq 0\}$; this language is not a tree-adjoining language.

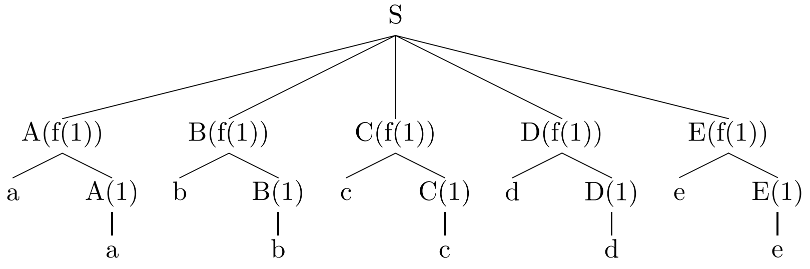

Figure 4. Derivation of the string aabbccddee in the DCG of Fig. 3.

The DCG framework is of interest in part because it so attractive to implement: indeed, DCGs are a built-in feature of the Prolog programming language (Pereira & Warren, 1980). The basis for such implementations is the concept of *unification*. We say that any terms t_1 and t_2 of $\mathcal{T}$ *unify* if there is a simultaneous substitution of terms for variables in t_1 and t_2 which make these expressions identical. If two terms unify, then there is a 'most general' unifier, which is unique up to renaming of variables. In a DCG-parser, when a non-terminal $a(u_1, \ldots, u_n)$ is expanded by the production (8), the most general unifier of the terms $a(u_1, \ldots, u_n)$ and $a(s_1, \ldots, s_n)$ is first computed; if this unifier exists, all variable bindings thus created are carried through to all the non-terminals $b_1(\bar{t}_1), \ldots, b_m(\bar{t}_m)$, which are then subject to expansion as before. Computing an explicit representation of the most general unifier of two terms is computationally expensive, because that representation is in general exponentially large in the size of the terms. However, determining *whether* two terms unify is much easier (Paterson & Wegman, 1978; de Champeaux, 1986):

Theorem 16. *The problem of determining whether two terms unify is in TIME*$(n + 1)$.

DCGs thus present an interesting object of study from a complexity-theoretic point of view. We have

Theorem 17. *The universal recognition problem for DCGs is undecidable. Indeed, there is a DCG G such that $L(G)$ is undecidable.*

Theorem 17 follows almost directly from Theorem 1, because the operation of any Turing machine M can easily be simulated using a DCG in which the values of variables are used to store configurations of M.

However, by imposing various reasonable constraints on DCGs, decidability can be restored. Let us say that a production is *consuming* if the right-hand side either consists of a single terminal or has length at least 2; and let us say that a DCG is *consuming* if all its productions are. For example, the production $\mathrm{a}(\mathrm{f}(x)) \rightarrow \mathrm{a}(x)$ is not consuming, because its right-hand side consists of a single non-terminal; on the other hand, the DCG of Fig. 3 is consuming. It is easy to show that, if a consuming DCG accepts a string σ of length n, the resulting parse-tree has at most $3n - 1$ nodes, so decidability in this case should not be a surprise. In fact, we have:

Theorem 18. *The universal recognition problem for consuming DCGs is NPTIME-complete. Indeed, there exists a consuming DCG G such that $L(G)$ is NPTIME-complete.*

The upper bound in Theorem 18 follows from the following observations. Given a consuming DCG G and a string σ of length n, we first guess a parse-tree featuring at most $3n - 1$ nodes. Each non-leaf node is labelled with the (uninstantiated) production of G responsible for generating it, and each leaf-node is labelled with a terminal, so as to form the string σ. (If the same production is used at more than one non-leaf node, new copies are made containing fresh variables.) We need only check that the terms in the copies of the productions at each node can be simultaneously unified in the obvious way. This check amounts to determining the unifiability of two (polynomially large) terms, and can be carried out in polynomial time by Theorem 16. The NPTIME-hardness of $L(G)$ for certain consuming DCGs G is easily shown by a simple reduction of the satisfiability problem for propositional logic clauses; the result then follows by Theorem 6.

Alternatively, we might say that a DCG is *function-free* if there are no function-symbols in its productions. (Thus, the DCG featured in Fig. 4 is not function-free, because several of its productions feature the function-symbol f.) We have:

Theorem 19. *The universal recognition problem for function-free DCGs is EXPTIME-complete. However, for any fixed function-free DCG G, $L(G)$ is in PTIME.*

Theorem 19 follows straightforwardly from the close connection between function-free DCGs and the logic programming language DATALOG. (See, e.g. Libkin (2004, Chapter 10), or Dantsin *et al.* (2001).) More generally, there is a close connection between DCGs on the one-hand and so-called fixed-point logics on the other, which allows standard results from complexity theory to be carried over to the study of DCGs. For example, Rounds (1988) describes two DCG-like grammar frameworks, one able to recognize all and only the languages in TIME(2^n), the other able to recognize all and only the languages in PTIME. Rounds shows that the second of his two grammar frameworks is at least expressive as that of TAG (and its equivalents), mentioned above.

A more traditional grammar framework generalizing CFGs is that of the context-sensitive grammars. A *context-sensitive grammar* (CSG) is like a CFG, except that the productions are now of the form $\alpha \to \beta$, where α and β are strings of symbols such that $|\alpha| \leq |\beta|$. These productions are interpreted as re-write rules, in much the same way as productions of a CFG. For comparison, note that, in a CFG, all productions have the form $A \to \beta$, where A is a non-terminal. Indeed, if we assume (which we may without essential loss of generality) that productions in CFGs have non-empty right-hand sides, the condition $|\alpha| \leq |\beta|$ is then trivially satisfied, whence CFGs are a special case of CSGs. Recalling the equivalence of languages and decision problems, it is routine to show that the class of context-sensitive languages is exactly the complexity class NSPACE(n). In fact, we have the following result concerning recognition complexity for context-sensitive languages.

Theorem 20. *The universal recognition problem for CSGs is PSPACE-complete. Indeed, there exists a CSG G such that $L(G)$ is PSPACE-complete.*

For a formal definition of context-sensitive grammars and a proof of Theorem 20, see Hopcroft & Ullman (1979, p. 223 and pp. 347 ff.). It was long conjectured that the complement of a context-sensitive language is itself a context-sensitive language. This conjecture was settled, positively, by Theorem 4, using the fact that the context-sensitive languages coincide with NSPACE(n).

All the grammar frameworks examined so far have precise formal definitions, which makes for a clear-cut complexity analysis. However, many mainstream grammar frameworks which aspire to describe natural languages are much less rigidly defined (and indeed much more liable to periodic revision); consequently, it is harder to provide definitive results about computational complexity. Transformational grammar is a case in point. Let us take a transformational grammar to consist of two components: a CFG generating a collection of phrase-structure trees—so-called *deep structures*—and a collection of *transformations* which map these deep structures to other phrase-structure trees—so-called *surface structures*. A string σ is *accepted* by G just in case σ can be read off the leaves of some surface-structure obtained in this way. Absent a formal specification of the sorts of transformations allowed in transformational grammar, it is impossible to determine the complexity of its recognition problem. However, analysing a version of Chomsky's *Aspects*-theory, Peters & Ritchie (1973) show the existence of transformational grammars which can recognize undecidable languages. Certainly, then, the universal recognition problem for transformational grammars (thus understood), is undecidable. Other analyses of grammar frameworks in the transformational tradition paint a picture of lower complexity, however. Thus, Berwick & Weinberg (1984), pp. 125 ff., analyse the complexity of *Government-Binding Gram-*

mars, a formalization of the approach taken in Chomsky (1981), and show that recognition complexity for such grammars is in the class PSPACE.

2.4 Model-theoretic semantics

Recent trends in linguistics—particularly within the transformational tradition—have shown a preference for specifying grammars not in terms of generative mechanisms, but rather, in terms of constraints to which sentence-structures are required to conform. On this view of grammar, a string σ is grammatical just in case it has a structure which satisfies those constraints. How can we determine the complexity of the recognition problem when grammars are presented in this way? The answer is to employ a formal language: this formal language must be powerful enough to express the constraints constituting the grammar in question, and yet not so powerful that working with it leads to undecidable problems.

Monadic second-order logic (MSO) is a formal language containing two sorts of variables: those ranging over objects (as in ordinary first-order logic), and those ranging over *sets* of objects. For the moment, let us suppose that the "objects" in question are positions in a string σ over an alphabet Σ. We confine ourselves to a language containing a unary atomic predicate s, for every $s \in \Sigma$, and binary predicates $\in$ and $\leq$. We now interpret these predicates over the set of positions in σ as follows (we adopt the convention of using lower-case letters for object-variables and upper-case letters for set-variables): $x \in X$ means "x is a member of X"; $x \leq y$ means "x is non-strictly to the left of y"; and $s(x)$ means "position x is filled with symbol s", for each $s \in \Sigma$. Formulas are built up from atomic formulas using Boolean connectives and quantifiers (over both sorts of variables) in the normal way. The standard semantics for these connectives then determine, for a given formula φ (with no free variables) and a given string σ, whether φ is true in σ. That is: any $\sigma \in \Sigma^*$ is a *structure* (in the logicians' sense) interpreting the above language.

On this view, we can think of an MSO-formula φ (with no free variables) as a *grammar*: a string σ is *accepted* by φ just in case φ is true in σ. The following result was proved by Büchi (60).

Theorem 21 (Büchi). *A language is recognized by an MSO-formula if and only if it is regular.*

Now, this approach to defining languages using formulas of MSO can be generalized in the following way. Suppose we take our variables to range, not over positions in strings, but over positions (nodes) in finite *trees*. (Think of the trees in question as phrase-structures of sentences.) And suppose we take our language to feature the binary predicates $\in$, $\lhd_1$ and $\lhd_2$, as well as unary predicates drawn from a finite set of labels. These predicates are then interpreted as follows: $x \in X$ again means "x is a member of X"; $x \lhd_1 y$ means "x is the mother of y"; $x \lhd_2 y$ means "x is a left sister of y"; and $s(x)$ means that x is labelled with s, for each label s. All other formulas are then interpreted according to the usual semantics of MSO. In this way, we can think of an MSO-formula φ (with no free variables) as licensing a set of labelled trees: namely, the trees in which φ is true. It was shown by Thatcher & Wright (1968) that the sets of trees (i.e. *tree-languages*) recognized in this way are—to within some additional labelling—the sets of trees generated by CFGs.

Indeed, one can interpret MSO-formulas over 'trees' of higher dimensions, obtaining grammar-frameworks of still greater expressive power. This approach to syntax is often referred to as *Model-theoretic syntax* (Rogers, 2003). Its appeal is partly due to the fact that MSO can express many relationships dear to linguists' hearts. For example, it is straightforward to write down a formula $\varphi_C(x, y)$ which is satisfied by nodes x and y in a tree just in case node x C-commands node y in that tree. Rogers *op. cit.* notes that some principles of Rizzi's theory of *Relativized Minimality* (Rizzi, 1990) can be expressed using formulas of the language sketched above. From a complexity-theoretic point of view, this approach is interesting because the problem of determining whether a formula of MSO is satisfiable over finite trees is decidable (Börger *et al.*, 1997, pp. 315 ff.):

Theorem 22. *The problem of determining the satisfiability of a formula of MSO over finite trees is decidable, but has non-elementary complexity.*

3 Complexity and semantics

Most linguistic theories are more than a criterion for defining a set of acceptable sentences: they also assign one or more levels of structure to those sentences which they do accept. The question then arises as to the computational complexity of recovering that structure.

Consider, for example, context-free grammars. Let G be a CFG. If $\sigma \in L(G)$, then G assigns to σ one or more phrase-structures representing the derivation of σ by the productions of G. It is easy to construct a CFG G for which there exists a sequence $\{\sigma_n\}_{n\in\mathbb{N}}$ of strings accepted by G, such that the length of σ_n is bounded above by some polynomial function of n, while the number of phrase-structures which G assigns to σ_n is bounded below by an exponential function of n. That is: the number of parses produced by a CFG G can grow exponentially. Nevertheless, the set of phrase-structures assigned to any string σ by G may always be compactly represented in the form of an acyclic directed graph, which can be expanded into a complete list of the phrase-structures in question; moreover, using a variant of the CYK or Earley algorithms, that compact representation may be computed in time $O(n^3m)$. (Trivially, listing all the represented phrase-structures will in general take exponential time.) For a general discussion on the relationship between the complexity of recognition and parsing, see Ruzzo (1979).

Arguably, determining the syntactic structure of a sentence is of little value unless we can use that structure to recover the sentence's meaning. The notion of meaning in general is too vague to admit of immediate formal analysis. However, we might sensibly begin with the more specific problem of recovering, at least for certain fragments of natural languages, *logical form*, in the sense of producing translations such as:

(9) Every boy loves some girl who admires him
$\forall x(\text{boy}(x) \rightarrow \exists y(\text{girl}(y) \wedge \text{admire}(y,x) \wedge \text{love}(x,y)))$.

The framework of CFGs (and indeed the other grammar frameworks mentioned above) can be modified to yield such logical forms. Approaches vary, but one popular technique is to associate with each vocabulary item an expression of the Simply-Typed λ-Calculus (STLC) representing its meaning, and to associate with each production a prescription for combining the meanings of the items in its right-hand side. In the following explanation, we assume basic familiarity with STLC; for an in-depth account, the reader is referred to Hindley & Seldin (1986, Chapter 13). A production in such a grammar has the form

$$A/\xi \rightarrow B_1/y_1 \ldots B_m/y_m,$$

where $y_1, \ldots, y_m$ are distinct variables, and ξ is an STLC-expression whose free variables are confined to $y_1, \ldots, y_n$. Such a production functions exactly as in an ordinary CFG, except that the meaning of the phrase A is computed by substituting the (already computed) meanings of the $B_1, \ldots, B_m$

for all occurrences of the corresponding variables $y_1, \ldots, y_m$ in ξ, and then β-reducing. This approach is, more or less, that championed by Montague (1974) (see Chapter 17, Section 2.1). For an accessible modern treatment, including a relatively non-technical explanation of the relevant aspects of higher-order logic, see Blackburn & Bos (2005). HARD REF

Consider, for example, the productions shown in Fig. 5. The underlying

$$\begin{array}{ll}
\text{IP}/y_1(y_2) \rightarrow \text{NP}/y_1\ \text{I}'/y_2 & \text{Det}/\lambda p\lambda q[\exists x(p(x) \wedge q(x))] \rightarrow \textsf{some} \\
\text{I}'/y_1 \rightarrow \textsf{is a}\ \ \text{N}'/y_1 & \text{Det}/\lambda p\lambda q[\forall x(p(x) \rightarrow q(x))] \rightarrow \textsf{every} \\
\text{I}'/\lambda x[\neg y_1(x)] \rightarrow \textsf{is not a}\ \ \text{N}'/y_1 & \text{Det}/\lambda p\lambda q[\forall x(p(x) \rightarrow \neg q(x))] \rightarrow \textsf{no} \\
\text{NP}/y_1 \rightarrow \text{PropN}/y_1 & \\
\text{NP}/y_1(y_2) \rightarrow \text{Det}/y_1\ \text{N}'/y_2 & \text{N/cynic} \rightarrow \textsf{cynic} \\
\text{N}'/y_1 \rightarrow \text{N}/y_1. & \text{N/philosopher} \rightarrow \textsf{philosopher} \\
 & \ldots \\
 & \\
 & \text{PropN}/\lambda p[p(\text{socrates})] \rightarrow \textsf{Socrates} \\
 & \text{PropN}/\lambda p[p(\text{diogenes})] \rightarrow \textsf{Diogenes} \\
 & \ldots
\end{array}$$

Figure 5. Semantically annotated CFG generating the language of the syllogistic.

CFG evidently recognizes the sentence Every cynic is a philosopher, via the parse-tree shown in Fig. 6. By computing the semantic values of each node in that tree, as shown, the (expected) first-order translation

$$\forall x(\text{cynic}(x) \wedge \rightarrow \text{philosopher}(x))$$

is eventually generated. In fact, the grammar of Fig. 5 recognizes the set of English sentences having the forms

$$\left\{ \begin{array}{c} \left\{ \begin{array}{l} \text{Every} \\ \text{Some} \\ \text{No} \end{array} \right\} L \\ S \end{array} \right\} \left\{ \begin{array}{l} \text{is a} \\ \text{is not a} \end{array} \right\} M,$$

where S is a proper noun, and L and M are common nouns, yielding, in each case, the expected translation into first-order logic. The question now arises: what is the computational complexity of recovering logical forms in this way?

The answer depends on how, exactly, logical forms are allowed to be represented. If the underlying grammar G is a CFG, then the CYK or Earley algorithms can again be modified to produce, in polynomial time, a compact representation of all meanings which G assigns to a given string σ, just as for parse-trees. However, these representations will not be β-reduced. That is, in order to produce ordinary logical translations such as (9), we need to com-

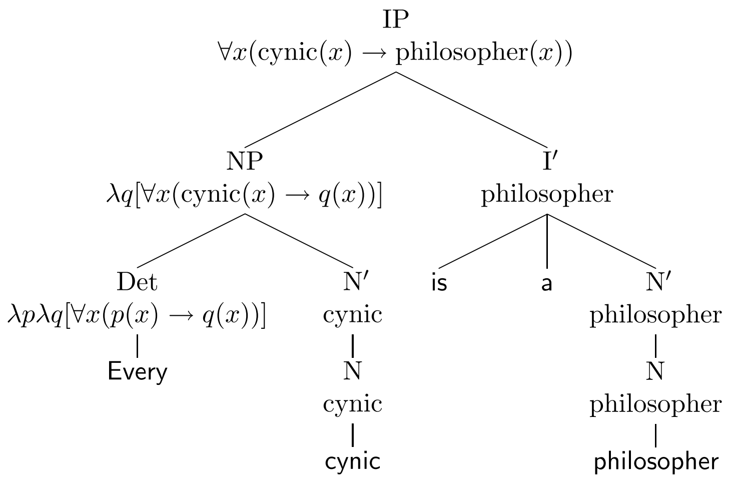
IP
$\forall x(\text{cynic}(x) \rightarrow \text{philosopher}(x))$
NP
$\lambda q[\forall x(\text{cynic}(x) \rightarrow q(x))]$
I′
philosopher
Det
$\lambda p \lambda q[\forall x(p(x) \rightarrow q(x))]$
Every
N′
cynic
N
cynic
cynic
is
a
N′
philosopher
N
philosopher
philosopher

Figure 6. Meaning derivation in a semantically annotated CFG.

pute the normal forms for the expressions which our parser yields. That these normal forms can be computed follows at once from Normalization Theorem for STLC, though the complexity of the relevant function is high (Statman, 1979):

Theorem 23. *The problem of deciding whether one expression in STLC is the normal form of another has non-elementary complexity.*

In practice, however, the normalization of semantic representations produced by realistic semantically annotated CFGs is never a problem.

4 Determining logical relationships between sentences

Computing anything is of little use if nothing is then done with the results. And while the uses to which humans put computed meanings may perhaps forever remain lost in the mists of psycholinguistics, Complexity Theory does have something to say about the more definite subject of determining *logical* relationships between sentences in natural language. That is the topic of this final section.

That sentences in natural language exhibit interesting logical relationships was recognized in antiquity. For example, the argument

$$(10)\qquad \begin{array}{l}\textsf{Every logician is a philosopher}\\ \textsf{Some stoic is a logician}\\ \underline{\textsf{No dentist is a philosopher}}\\ \textsf{Some stoic is not a dentist,}\end{array}$$

is evidently valid: every possible situation in which the premises are true is one in which the conclusion is true. Likewise valid, but less evidently so, is the argument

$$(11)\qquad \begin{array}{l}\textsf{Every sceptic recommends every sceptic to every cynic}\\ \textsf{No sceptic recommends any stoic who hates any cynic}\\ \quad\textsf{to any philosopher}\\ \textsf{Diogenes is a cynic who every sceptic hates}\\ \underline{\textsf{Every cynic is a philosopher}}\\ \textsf{No stoic is a sceptic.}\end{array}$$

Observe that Argument (11) uses a wider variety of grammatical constructions than Argument (10), specifically: transitive and ditransitive verbs, as well as relative-clauses. The question therefore arises as to how the difficulty of determining logical relationships between sentences in naturally delineated fragments of natural languages depends on the grammatical resources included in those fragments. Are ditransitive verbs really harder than transitive verbs? Passives harder than actives? How much extra effort is required to deal with relative clauses (either subject relatives or object relatives)? Is 'donkey'-anaphora more computationally intensive than other forms of bound-variable anaphora? And so on.

Consider the grammar of Fig. 5, which, as we saw in Section 3, yields the language of the traditional syllogistic. In particular, this grammar recognizes all the sentences in Argument (10), and translates that argument to the first-order sequent

$$\begin{array}{l}\forall x(\text{logician}(x) \rightarrow \text{philosopher}(x))\\ \exists x(\text{stoic}(x) \wedge \text{logician}(x))\\ \underline{\forall x(\text{dentist}(x) \rightarrow \neg\text{philosopher}(x))}\\ \exists x(\text{stoic}(x) \wedge \neg\text{dentist}(x)).\end{array}$$

Since the primary form-determining element in this fragment of English is the copula, we refer to it as *Cop*. With translations into first-order logic at our disposal, we can now formally characterize a notion of validity in this fragment. Specifically, we take an argument in the fragment Cop to be *valid* just in case the first-order sequent into which it is translated is valid according to the semantics of first-order logic. Likewise, we take a set of sentences in Cop to be *satisfiable* just in case the set of formulas to which they are translated is satisfiable according to the semantics of first-order logic.

Thus, the fragment Cop is more than a mere set of strings (the grammatical sentences): it is a set of strings together with associated logical concepts of validity and satisfiability. In particular, we may pose the satisfiability problem for Cop: given a set E of sentences in Cop, determine whether E is satisfiable. Furthermore, since every sentence in Cop is logically equivalent to the negation of some other, satisfiability and validity are dual notions, in the familiar sense: an argument is valid just in case its premises together with the negation of its conclusion are unsatisfiable. Hence, the complexity of the validity problem for Cop can be determined immediately from the complexity of the satisfiability problem.

It is routine to show that determining the satisfiability of a collection of sentences in the fragment Cop is essentially the same as the problem of determining the satisfiability of a collection of propositional clauses each of which contains at most two literals. Recalling Theorem 8, we have:

Theorem 24. *The problem of determining the satisfiability of a set of sentences in Cop is NLOGSPACE-complete.*

It follows of course that the problem of determining the validity of an argument in Cop is also NLOGSPACE-complete, by Theorem 4. This confirms our subjective impression that this problem is nearly trivial.

What happens if we expand the fragment Cop? Let us define the fragment *Cop+TV* to be the set of sentences recognized by the productions of Fig. 5 together with those of Fig. 7. (We have simplified the treatment by ignoring verb-inflections and negative-polarity determiners; these simplifications are not computationally significant.) It is easy to see that this fragment contains

$\mathrm{I}'/y_1 \rightarrow \mathrm{VP}/y_1$	$\mathrm{TV}/\lambda s \lambda x[s(\lambda y[\mathrm{admire}(x, y)])]$
$\mathrm{I}'/y_1 \rightarrow \mathrm{NegP}/y_1$	$\rightarrow$ admires
$\mathrm{NegP}/\lambda x[\neg y_1(x)] \rightarrow \mathrm{Neg}\ \mathrm{VP}/y_1$	$\mathrm{TV}/\lambda s \lambda x[s(\lambda y[\mathrm{despise}(x, y)])]$
$\mathrm{VP}/y_1(y_2) \rightarrow \mathrm{TV}/y_1\ \mathrm{NP}/y_2$	$\rightarrow$ despises
$\mathrm{Neg} \rightarrow$ does not	...

Figure 7. Productions for extending the syllogistic with transitive verbs.

the following sentence, and translates it to the indicated first-order formula.

(12) Every stoic hates every sceptic
$\forall x(\mathrm{stoic}(x) \rightarrow \forall y(\mathrm{sceptic}(y) \rightarrow \mathrm{hate}(x, y)))$.

We need to address the issue of scope ambiguities in the context of Cop+TV. There are two possibilities here: either we can resolve these ambiguities by fiat, taking subjects always to outscope objects; or we can augment the language with some form of marking to indicate quantifier scope. For simplicity, we choose the former course (though the latter would lead to essentially the same complexity results). Similarly, let *Cop+TV+DTV* be the fragment which extends Cop with both transitive and ditransitive verbs. (Writing the required productions is completely routine.) Thus, Cop+TV+DTV contains the following sentence, and translates it to the indicated first-order formula.

(13) No stoic recommends every sceptic to some cynic
$\forall x(\text{stoic}(x) \rightarrow \neg\forall y(\text{sceptic}(y) \rightarrow \exists z(\text{cynic}(z) \wedge \text{recommend}(x, y, z))))$.

Again, we take subjects to outscope direct objects, and direct objects to outscope indirect objects.

Is inference in these larger fragments more complex? The following two results (substantially) answer this question.

Theorem 25. *The problem of determining the satisfiability of a set of sentences in Cop+TV is in NLOGSPACE-complete.*

Theorem 26. *The problem of determining the satisfiability of a set of sentences in Cop+TV+DTV is in PTIME.*

The proofs of these theorems are more elaborate than for Cop; we refer the reader to Pratt-Hartmann & Moss (2009) and Pratt-Hartmann & Third (2006), respectively.

Returning to the fragment Cop, what happens if we now add relative clauses? Thus, for example, we have the valid argument

Every philosopher who is not a stoic is an epicurean
No epicurean is a beekeeper
No stoic is a beekeeper
―――――――――――――
No philosopher is a beekeeper,

It is straightforward to write a semantically annotated context-free grammar accepting such sentences, and generating the obvious semantics. Let us call the resulting fragment of English Cop+Rel (see Pratt-Hartmann (2004) for a formal definition). The first-order formulas into which Cop+Rel-sentences are translated all have one variable—that is to say, they lie within the 1-variable fragment of first-order logic. The satisfiability problem for this fragment is essentially the same as that for clauses of the propositional calculus. Thus, from Theorem 6:

Theorem 27. *The problem of determining the satisfiability of a set of sentences in Cop+Rel is NPTIME-complete.*

On the other hand, the fragment Cop+Rel+TV+DTV recognizes all the sentences in Argument (11), and translates them into the first-order sequent

$$\begin{array}{l}
\forall x(\text{sceptic}(x) \rightarrow \forall y(\text{sceptic}(y) \rightarrow \forall z(\text{cynic}(z) \rightarrow \text{recommends}(x, y, z))))\\
\forall x(\text{sceptic}(x) \rightarrow \neg\exists y(\text{stoic}(y) \wedge \exists z(\text{cynic}(w) \wedge \text{hate}(y, w))\wedge\\
\qquad\qquad\qquad\qquad \exists z(\text{philosopher}(z) \wedge \text{recommends}(x, y, z))))\\
\text{cynic}(\text{diogenes}) \wedge \forall x(\text{sceptic}(x) \rightarrow \text{hate}(x, \text{diogenes}))\\
\underline{\forall x(\text{cynic}(x) \rightarrow \text{philosopher}(x))}\\
\forall x(\text{stoic}(x) \rightarrow \neg\text{sceptic}(x)).
\end{array}$$

Again, we have the question: does adding transitive and ditranstive verbs to Cop+Rel lead to an increase in complexity? This time, the answer is yes.

Theorem 28. *The problem of determining the satisfiability of a set of sentences in Cop+Rel+TV is EXPTIME-complete.*

Theorem 29. *The problem of determining the satisfiability of a set of sentences in Cop+Rel+TV+DTV is NEXPTIME-complete.*

Theorem 29 confirms our subjective impression that determining the validity of Argument (11) is harder than determining the validity of Argument (10). For proofs of the above theorems, see Pratt-Hartmann (2004); Pratt-Hartmann & Third (2006).

A remark is in order at this point to correct a false impression that the foregoing discussion may have created. As we have observed: the complexity of determining entailments within a fragment of a natural language evidently depends on the constructions made available by the syntax of that fragment. However, it also depends, of course, on the presence in the lexicon of words with a 'logical' character. Consider, for example, the effect of expanding the fragments Cop and Cop+TV with *numerical* determiners, yielding sentences such as

(14) At least 13 artists are beekeepers

in the former case, and

(15) At most 5 carpenters admire at most 4 dentists

in the latter. Calling the resulting fragments Cop+Num and Cop+TV+Num, we obtain the following results (Pratt-Hartmann, 2008):

Theorem 30. *The problem of determining the satisfiability of a set of sentences in Cop+Num is NPTIME-complete; the problem of determining the satisfiability of a set of sentences in Cop+TV+Num is NEXPTIME-complete.*

Thus, the complexity-theoretic impact of such numerical expressions is dramatic.

Finally, we consider the complexity-theoretic consequences of adding bound-variable anaphora to our fragments. Consider the sentences

No artist admires any beekeeper who does not admire himself
No artist admires any beekeeper who does not admire him.

It is routine to add grammar rules to Cop+Rel+TV producing the conventional translations into first-order logic:

$$\forall x(\text{artist}(x) \to \forall y(\text{beekeeper}(y) \wedge \neg\text{admire}(y,y) \to \text{admire}(x,y)))$$
$$\forall x(\text{artist}(x) \to \forall y(\text{beekeeper}(y) \wedge \neg\text{admire}(y,x) \to \text{admire}(x,y))).$$

For such anaphoric fragments, two further issues regarding the first-order translations arise. First, we assume the (standard) universal interpretation of 'donkey-sentences'

Every farmer who owns a donkey beats it
$$\forall x \forall y(\text{farmer}(x) \wedge \text{donkey}(y) \wedge \text{own}(x,y) \to \text{beat}(x,y)).$$

Second, we must decide how to treat anaphoric ambiguities. The sentence

(16) Every sceptic who admires a cynic despises every stoic who hates him,

has two interpretations:

$$\forall x(\text{sceptic}(x) \wedge \exists y(\text{cynic}(y) \wedge \text{admire}(x,y)) \to \forall z(\text{stoic}(z) \wedge \text{hate}(z,x) \to \text{despise}(x,z))) \tag{17}$$

$$\forall x \forall y(\text{sceptic}(x) \wedge \text{cynic}(y) \wedge \text{admire}(x,y) \to \forall z(\text{stoic}(z) \wedge \text{hate}(z,y) \to \text{despise}(x,z))). \tag{18}$$

according as the pronoun him takes as antecedent the NP headed by sceptic or the NP headed by cynic. (The NP headed by stoic is not available as a pronoun antecedent here.)

Note that, in the (standard) phrase-structure tree for this sentence, the NP headed by sceptic is closer to the pronoun than is the NP headed by cynic. This observation suggests making the artificial stipulation that *pronouns must take their closest allowed antecedents*. Here, *closest* means 'closest measured along edges of the phrase-structure' and *allowed* means 'allowed by the principles of binding theory'. (We ignore case and gender agreement.) Thus, under this stipulation, Sentence (16) has only the reading (17). Let the resulting fragment of English, with the stipulation of closest available pronomial antecedents, be called Cop+Rel+TV+RA ('RA' for *restricted anaphora*).

Formula (17) can be equivalently written

$$\forall x(\text{sceptic}(x) \wedge \exists y(\text{cynic}(y) \wedge \text{admire}(x,y)) \to \forall y(\text{stoic}(y) \wedge \text{hate}(y,x) \to \text{despise}(x,y))),$$

with the variable z replaced by y. The resulting formula has only two variables. Indeed, it can be shown that every sentence of Cop+Rel+TV+RA translates into a formula in the 2-variable fragment of first-order logic. The satisfiability problem for this fragment is known to be NEXPTIME-complete (Börger *et al.*, 1997, Chapter 8). Moreover, Cop+Rel+TV+RA can easily be shown to encode a NEXPTIME-hard problem. Hence, we have:

Theorem 31. *The problem of determining the satisfiability of a set of sentences in Cop+Rel+TV+RA is NEXPTIME-complete.*

We mention in passing that the reduction of NEXPTIME-hard problems to satisfiability for sets of Cop+Rel+TV+RA-sentences does not require the use of sentences featuring donkey-anaphora. However awkward such sentences may be for the smooth-running of formal semantics, they do not lead to more complex inferential problems.

The restriction that pronouns take their closest possible antecedents is essential to the complexity bound of Theorem 31. As an alternative treatment of anaphoric ambiguitiy, we might augment the sentences of Cop+Rel+TV+RA with indices indicating antecedents in the normal way. Thus, for example, the sentence

Every sceptic$_1$ who admires a cynic$_2$ despises every stoic$_3$ who hates him$_2$

would have (18) as its only reading. Let the resulting fragment be denoted by Cop+Rel+TV+GA ('GA' for *general anaphora*). It is possible to show:

Theorem 32. *The problem of determining the satisfiability of a set of sentences in Cop+Rel+ TV+GA is not decidable.*

It seems clear that many more results of the kind outlined in this section await discovery.

See also: Chapter 1, Formal Language Theory;
Chapter 4, Theory of Parsing;
Chapter 17, Computational Semantics.

References

Berwick, R. C. & Amy S. Weinberg (1984), *The Grammatical Basis of Linguistic Performance: Language Use and Acquisition*, MIT Press, Cambridge, MA.

Blackburn, Patrick & Johan Bos (2005), *Representation and Inference for Natural Language. A First Course in Computational Semantics*, CSLI.

Börger, Egon, Erich Grädel, & Yuri Gurevich (1997), *The Classical Decision Problem*, Perspectives in Mathematical Logic, Springer-Verlag, Berlin.

Büchi, J.R. (60), Weak second-order arithmetic and finite automata, *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik* 6:66–92.

Carpenter, Bob (1997), *Type-logical Semantics*, MIT Press, Cambridge, MA.

de Champeaux, Dennis (1986), About the Paterson-Wegman linear unification algorithm, *Journal of Computer and System Sciences* 32:79–90.

Chomsky, Noam (1981), *Lectures on Government and Binding*, Foris Publications, Dordrecht.

Cook, S.A. (1971), The complexity of theorem-proving procedures, in *Proceedings of the 3rd IEEE Symposium on the Theory of Computation*, IEEE Press, (151–158).

Dantsin, Evgeny, Thomas Eiter, Georg Gottlob, & Andrei Voronkov (2001), Complexity and expressive power of logic programming, *ACM Computing Surveys* 33(3):374 – 425.

Earley, Jay (1970), An efficient context-free parsing algorithm, *Communications of the ACM* 13(2):94–102.

Hindley, J. Roger & Jonathan P. Seldin (1986), *Introduction to Combinators and λ-Calculus*, Cambidge University Press, Cambridge.

Hopcroft, John E. & Jeffrey D. Ullman (eds.) (1979), *Introduction to Automata Theory, Languages and Computation*, Addison Wesley, Reading, MA.

Immerman, N. (1988), Nondeterministic space is closed under complement, *SIAM Journal on Computing* 17:935–938.

Jones, Neil D. & William T. Laaser (1977), Complete problems for deterministic polynomial time, *Theoretical Computer Science* 3(1):105–118.

Kozen, Dexter C. (2006), *Theory of Computation*, Springer Verlag, London.

Lambek, Joachim (1958), The mathematics of sentence structure, *American Mathematical Monthly* 65(3):154–170.

Libkin, Leonid (2004), *Elements of Finite Model Theory*, Springer, Berlin.

Montague, Richard (1974), The proper treatment of quantification in ordinary English, in R. Thomason (ed.), *Formal Philosophy*, Yale University Press, New Haven, CT, (247–70).

Nepomnyashchii, V. A. (1975), Spatial complexity of recognition of context-free languages, *Cybernetics and Systems Analysis* 11(5):736–741.

Papadimitriou, Christos H. (1994), *Computational Complexity*, Addison-Wesley, Reading, MA.

Paterson, M.S. & M.N. Wegman (1978), Linear unification, *Journal of Computer and System Sciences* 16:158–167.

Pentus, M. (1993), Lambek grammars are context free, in *Proceedings, 8th Annual Symposium on Logic in Computer Science*, IEEE Press, (429–433).

Pentus, M. (1994), Language completeness of the Lambek calculus, in *Proceedings, 9th Annual Symposium on Logic in Computer Science*, IEEE Press, (487–496).

Pentus, Mati (1997), Product-free Lambek calculus and context-free grammars, *Journal of Symbolic Logic* 62(2):648–660.

Pentus, Mati (2006), Lambek calculus is NP-complete, *Theoretical Computer Science* 357(1–3):186–201.

Pereira, Fernando C. N. & David H. D. Warren (1980), Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks, *Artificial Intelligence* 13(3):231–278.

Peters, P. Stanley, Jr. & R.W. Ritchie (1973), On the generative power of transformational grammars, *Information Sciences* 6:49–83.

Pratt-Hartmann, Ian (2004), Fragments of language, *Journal of Logic, Language and Information* 13:207–223.

Pratt-Hartmann, Ian (2008), On the computational complexity of the numerically definite syllogistic and related logics, *Bulletin of Symbolic Logic* 14(1):1–28.

Pratt-Hartmann, Ian & Lawrence S. Moss (2009), Logics for the relational syllogistic, *Review of Symbolic Logic* Forthcoming.

Pratt-Hartmann, Ian & Allan Third (2006), More fragments of language, *Notre Dame Journal of Formal Logic* 47(2):151–177.

Ritchie, R.W. & F.N. Springsteel (1972), Language recognition by marking automata, *Information and Control* 20(4):313–330.

Rizzi, Luigi (1990), *Relativized Minimality*, MIT Press, Cambridge, MA.

Rogers, James (2003), Syntactic structures as multi-dimensional trees, *Journal of Language and Computation* 1(3–4):265–305.

Rounds, W. (1988), LFP: A logic for linguistic descriptions and an analysis of its complexity, *Computational Linguistics* 14(4):1–9.

Ruzzo, Walter L. (1979), On the complexity of general context-free language parsing and recognition, in H. A. Maurer (ed.), *Automata, Languages and Programming, Sixth Colloquium*, Springer Verlag, Berlin, volume vol. 71. of *Lecture Notes in Computer Science*, (479–488).

Savitch, W. (1970), Relationship between nondeterministic and deterministic tape complexities, *Journal of Computer and System Sciences* 4:177–192.

Schabes, Yves (1994), Left-to-right parsing in lexicalized tree-adjoining grammars, *Computational Intelligence* 10(4):506–524.

Statman, Richard (1979), The typed λ-calculus is not elementary recursive, *Theoretical Computer Science* 9:73–81.

Stockmeyer, L. & A.R. Meyer (1973), Word problems requiring exponential time: a preliminary report, in *Proceedings of the fifth annual ACM symposium on Theory of computing*, ACM Digital Library, (1 – 9).

Thatcher, J. & J. Wright (1968), Generalized finite automata theory with an application to a decision problem of second-order logic, *Mathematical systems theory* 2:57–81.

Turing, Alan (1936–7), On computable numbers, with an application to the *Entscheidungsproblem*, *Proceedings of the London Mathematical Society* (*Series 2*) 42:230–265.

Vijay-Shanker, K. & David Weir (1994), The equivalence of four extensions of context-free grammars, *Mathematical Systems Theory* 27:511–545.

Younger, Daniel H. (1967), Recognition and parsing of context-free languages in time n^3, *Information and Control* 10:189–208.

Yu, Sheng (1997), Regular languages, in G. Rozenberg & A. Salomaa (eds.), *Handbook of Formal Languages*, Springer, Berlin, volume 1, chapter 2, (41–110).