Chapter 1

Introduction

What makes a language a *natural* language? One long-standing and fruitful approach holds that a language is natural just in case it is learnable. Antedating this focus on learnability, though, was a mathematically grounded taxonomy that sought to classify the power of grammatical theories via the string sets (*languages*) the theories could generate—their *weak generative capacity*. Weak generative capacity analysis can sometimes identify inadequate grammatical theories: for example, since most linguists would say that any natural grammar must be able to generate sentences of unbounded length, we can disqualify any grammatical system that generates only finite languages. For the most part, formal grammatical analysis has remained firmly wedded to weak generative capacity and the Chomsky hierarchy of finite-state, context-free, context-sensitive, and type-0 languages. Linguists still quarrel about whether the set of English sentences (regarded just as a set of strings) is context-free or not, or whether one or another formalism can generate the strictly context-sensitive string pattern xx.

This book aims to update that analytic tradition by using a more recent, powerful, and refined classification tool of modern computer science: computational complexity theory. It explains what complexity theory is and how to use it to analyze several current grammatical formalisms, ranging from lexical-functional grammar, to morphological analysis systems, to generalized phrase structure grammar; and it outlines its strengths and limits.[1]

[1] Other recent formal approaches also seek alternatives to weak generative capacity analysis. For example, Rounds, Manaster-Ramer, and Friedman (1986) propose that natural language grammars cannot be "too large" in the sense that the number of sentences they can generate must be substantially larger than the number of nonterminals they contain. This formal constraint, plainly intertwined with the issues of succinctness and learnability

Complexity theory studies the computational resources—usually time and memory space—needed to solve particular problems, abstracting away from the details of the algorithm and machine used to solve them. It gives us robust classification schemes—*complexity classes*—telling us that certain problems are likely or certain to be computationally tractable or intractable—where, roughly speaking, "tractable" means always solvable in a reasonable amount of time and/or space on an ordinary computer. It works by comparing new problems to problems already known to be tractable or intractable. (Section 1.2 below says more, still informally, about what we mean by a tractable or intractable problem and how we show a new problem to be tractable or intractable. Chapter 2 gives a more formal account.)

Importantly, this classification holds regardless of what algorithm we use or how many top-notch programmers we hire—in other words, a hard problem can't be switched into an easier complexity class by using a clever algorithm—and it holds regardless of whether we use a modest PC or a much faster mainframe computer. Abstracting away from computer and algorithm details seems especially apt for consideration of linguistic processing, since for the most part we don't know what algorithm or computing machinery the brain uses, but we *do* know—with the linguist's help—something about the abstract natural language problems that language processing mechanisms must grapple with.[2]

1.1 Complexity Theory as a Theoretical Probe

If we're investigating the processing difficulty of grammatical problems, complexity theory offers four main advantages over weak generative capacity analysis:

- It is *more direct* and *more refined.* If we want to know something about how long it takes to process a grammatical problem on a computer, then that's what complexity theory tells us, without going through any intermediate steps linking weak generative capacity to time or space use.

so dear to the linguist's heart, may also yield interesting results, yet is quite distinct from the results of conventional complexity theory.

[2]Given complexity theory's focus on "ordinary" computers, those interested in the impact of parallel computation on our results should consult section 1.4.5 at the end of this chapter and section 2.4 in the next.

Further, we can set up many more than just the four rough categories of the Chomsky hierarchy—and that's useful for probing the complexity of systems that don't fit neatly into the finite-state–context-free–context-sensitive picture. (See section 1.2 and chapters 2 and 8 for examples.)

- It is *more accurate*. Weak generative capacity results can give a misleading picture of processing difficulty. For example, just because a grammatical system uses finite-state machinery does *not* guarantee that it can be efficiently processed; chapter 5 shows why. Similarly, strictly context-free generative power does *not* guarantee efficient parsability (see chapters 7 and 8).
- It is *more robust*. We have already mentioned the theory's independence from details of computer model and algorithm. But it can also tell us something about the beneficial effects of parallel computation, if any, without having to wait to buy a parallel computer (see sections 1.4 and 2.4).
- It is *more helpful*. Since complexity analysis can tell us *why* a grammatical formalism is too complex, it can also sometimes tell us *how* to make it less complex. Chapters 8 and 9 show how to use complexity theory to revise generalized phrase structure grammar so as to make it much more tractable (though still potentially difficult).

But some might question why we need this computational armament at all. Isn't it enough just to pick grammatical machinery that has more than enough power to describe natural languages, and and then go out and use it? One reason we need help from complexity theory and other tools is that using a powerful metalanguage to express grammars—whether it's drawn from mathematics or plain English—doesn't give us much guidance toward writing down only *natural* grammars instead of unwittingly composing unnatural ones.

To take a standard linguistic example, suppose we use the language of context-free grammars as our descriptive machinery. Then we can write down natural grammar rules for English like these:

$$VP \rightarrow Verb\ NP \quad PP \rightarrow Prep\ NP$$

but we can also write down the unnatural rules,

$$VP \rightarrow Noun\ NP \quad PP \rightarrow VP\ Noun\ PP$$

In this case, the generality of the machinery blinds us to some of the natural structure of the problem—we miss the fact that every phrase of type X has

a distinguished head of the same type, with verb phrases headed by verbs, prepositional phrases by prepositions, and so forth (as expressed in many modern frameworks by $\overline{X}$ theory). For linguistic purposes, a better framework would yield only the natural grammars, steering us clear of such errors.

We should like to enlist complexity theory in this same cause. Implicitly, our faith in complexity analysis boils down to this: complexity analysis tells us *why* problems are easy or hard to solve, hence giving us insight into the *information processing structure* of grammatical systems. It can help pinpoint the exact way in which our formalized systems seem to allow too much latitude—for instance, identifying the parts of our apparatus that let us describe languages that seem more difficult to process than natural languages. Especially deserving of closer scrutiny are formal devices that can express problems requiring blind, exhaustive, and computationally intractable search for their solution. Informally, such computationally difficult problems don't have any special structure that would support an efficient solution algorithm, so there's little choice but brute force, trying every possible answer combination until we find one that works. Thus, it's particularly important to examine features of a framework that allow such problems to be encoded—making sure there's not some special structure to the natural problem that's been missed in the formalism.

In fact, problems that require combinatorial search might well be characterized as *unnaturally hard problems.*[3] While there is no *a priori* reason why a theory of grammatical competence must guarantee efficient processing, there is every reason to believe that natural language has an intricate computational structure that is not reflected in combinatorial search methods. Thus, a formalized problem that requires such search probably leaves unmentioned some constraints of the natural problem. We'll argue in chapter 6 that the best grammatical framework will sometimes leave a residue of worst-case computational difficulty, so hard problems don't automatically indicate an overly general formalism; like other tools, complexity results should be interpreted intelligently, in the light of other evidence. But even when the framework must allow hard problems, we believe the intractability still warns that we may have missed some of the particular structure of natural language—and it can guide us toward what and where. *Performance methods* may well assume special properties of natural language beyond those that are guaranteed by the grammatical formalism, hence succeeding when the special

[3]Such problems are difficult even if one allows a physically realistic amount of parallel computation; see section 1.4.5.

properties hold, but failing in harder situations where they do not. In chapters 5 and 6 we explore such a possibility (among other topics), sketching a processing method that assumes natural problems typically have a more modular and local structure than computationally difficult problems.

To consider a simple example here, chapter 5 studies the dictionary-retrieval component of a natural language processing system: for instance, a surface form like `tries` may be recovered as the underlying form `try+s`. We can solve this abstract problem by modeling possible spelling changes with a set of finite-state transducers that map between surface and underlying forms. However, this *two-level model* can demand exhaustive search. For example, when processing the character sequence "`s p i...`" left-to-right, the two-level system must decide whether or not to change the surface "`i`" to an underlying "`y`", guessing that underlying word is something like `spy+s`. But this guess could go awry because the underlying word could be `spiel`, and when we look closely at the range of problems allowed by the two-level model, full combinatorial search—guessing and backtracking—seems to be required. In fact, chapter 5 shows that the backtracking isn't just symptomatic of a bad algorithm for implementing this model; in the general case, the two-level model is computationally intractable, independent of algorithm and computer design.

In practice, two-level processing for natural languages does involve search, but less search than we find when we run the reduction that demonstrates possible intractability. We should therefore ask whether there is something special about the structure of the natural problems that makes them more manageable than the formal model would suggest—something that the model fails to capture, hence allowing unnaturally difficult situations to arise. Chapter 6 suggests that this might be so, for preliminary results indicate that a weaker but noncombinatorial processing method—constraint propagation—may suffice for natural spelling-change systems. The constraint-propagation method assumes natural spelling changes have a local and separable character that is not implied in the two-level model.

If our approach is on the right track, then a grammatical formalism that in effect poses brute-force problems should make us suspicious; complexity analysis gives us reason to suspect that the special structure of the human linguistic system is not being exploited. Then complexity analysis may help pinpoint the computational sore spots that deserve special attention, suggesting additional restrictions for the grammatical systems or alternative, approx-

imate solution methods. Chapter 4 applies complexity-theory diagnostic aids to help repair lexical-functional grammar; as we mentioned earlier, chapters 8 and 9 do the same for generalized phrase structure grammar.

But when linguistic scrutiny bears out the basic validity of the formal system—when the grammatically defined natural problems are just plain hard—then the complexity diagnosis suggests where to seek performance constraints. Chapter 3 gives an example based on a simple grammatical system that contains just the machinery of *agreement* (like the agreement between a noun phrase subject and a verb in English) and *lexical ambiguity* (in English, a word such as *kiss* can be either a noun or a verb). This system is computationally intractable, but in a way that's roughly reflected in human performance: sentences that lack surface information of categorial features are hard to process, as we see from the sentence BUFFALO BUFFALO BUFFALO. We mention this example again in chapters 3 and 6.

Finally, if a grammatical problem is easy, then complexity analysis again can tell us why that's so, based on the structure of the problem rather than the particular algorithms we've picked for solving the problem; it can help tell us why our fast algorithms work fast. In a similar way, it can help us recognize systems in which fast processing is founded on unrealistic restrictions (for instance, perhaps a prohibition against lexical ambiguity).

To give the reader a further glimpse of our methods and results, the rest of this chapter quickly and informally surveys what complexity theory is about, how we apply it to actual grammatical systems, and what its limits are. The next chapter takes a more detailed and thorough look at the connection between complexity theory and natural language.

Section 1.2 introduces a few core concepts from complexity theory: it identifies the class $\mathcal{P}$ as the class of tractable problems, includes the hardest problems of the class $\mathcal{NP}$ in the class of intractable problems, and briefly discusses how we can use representative problems in each class to tell us something about the complexity of new problems. Section 1.3 illustrates how we apply complexity theory techniques to grammatical systems by analyzing an artificially simplified grammatical formalism. Section 1.4 briefly reviews the virtues and limits of complexity analysis for cognitive science, addressing questions about idealization, compilation effects, and parallel computation. Section 1.5 concludes the chapter with an outline of the rest of the book, highlighting our main results.

1.2 What Complexity Theory is About

We know that some problems can be solved quickly on ordinary computers, while others cannot be. Complexity theory captures our intuitions by defining classes that lump together entire sets of problems that are easy to solve or not.

1.2.1 Problem vs. algorithm complexity

We have said several times that we aim to study problem complexity, not algorithm complexity, because it's possible—even easy—to write a slow algorithm for an easy problem, and this could be seriously misleading. So let us drive home this distinction early on, before moving on to problem complexity analysis itself.

Consider the problem of searching a list of alphabetically sorted names to retrieve a particular one. Many algorithms solve this problem, but some of them are more efficient than others. For example, if we're looking for "Bloomfield," we could simply scan through our list starting with the "A" words, comparing the name we want against the names we see until we hit the right name. In the worst case we might have to search all the way through to the end to find the one we're looking for—for a list of n names, this would be at worst proportional to n basic comparisons.

This smacks of brute-force search, though it's certainly not the exponential search we're usually referring to when we mention brute-force methods. Another algorithm does much better by exploiting the structure of the problem. If we look at the middle name in our list—say, "Jespersen"—we can compare it to our target name. If that name ranks alphabetically below our target, then we repeat our procedure by taking just the top half of our list of names, finding the middle in that new halved list, and comparing it against our target. (If the name ranks alphabetically above our target, then we repeat our search in the bottom half of the list.) It's easy to see that in the worst case this *binary search* algorithm makes fewer comparisons—we can keep halving things only so far before we get a lone name in each half, and the number of splits is roughly proportional to $\log_2 n$. This second algorithm exploits the special structure of our alphabetically sorted list to work better than blind search. In this case then, complexity lies in the algorithm, not in the problem.

1.2.2 Easy and hard problems; P and NP

With the algorithm–problem distinction behind us, we can move on to look at problem complexity. Easy-to-solve problems include alphabetical sorting, finite-state parsing, and context-free language recognition, among others. For example, context-free language recognition takes at worst time proportional to $|x|^3$, where $|x|$ is the number of words in the sentence, if we use a standard context-free recognition algorithm like CKY (Hopcroft and Ullman 1979). Indeed, all of the above-mentioned problems take time proportional to n, or $\log n$, $n \log n$, or n^3, where n measures the "size" of the problem to solve. More generally, all such problems take at most some *polynomial* amount of time to solve on a computer—at most time proportional to n^j, for some integer j. Complexity theory dubs this the class P: the class of problems solvable (by some algorithm or other) in polynomial time on an ordinary computer. (Recall that an *algorithm's* complexity is to be distinguished from a *problem's* complexity: it's possible to write a bad alphabetic sorting algorithm that takes more than polynomial time, yet the sorting problem is in P. Significantly, it's not possible to write a preternaturally good algorithm that takes *less* time in the worst case than the complexity of the problem would indicate.)

Still other problems seem to take longer to solve no matter what algorithm one tries. Consider the following example, known as Satisfiability or SAT: Given an arbitrary Boolean formula like the following:

$$(x \vee \overline{y} \vee \overline{z}) \wedge (y \vee z \vee u) \wedge (x \vee z \vee \overline{u}) \wedge (\overline{x} \vee y \vee u)$$

is there an assignment of **true** and **false** to the variables such that the whole expression is true? In this case we say that the formula is *satisfiable*, otherwise, *unsatisfiable*. Note that $\wedge$ is logical *and* while $\vee$ is logical *or*, so every clause in parentheses has to have at least one literal that is true, where $\overline{x}$ is true if x is false, and vice-versa.[4]

[4] We assume that satisfiability formulas are in *conjunctive normal form*, stated as a collection of clauses each of which contains any number of negated or unnegated variables (so-called *literals*) in the form x or $\overline{x}$. Each clause must contain at least one literal that is true. A slightly more general version of Boolean expressions is sometimes used, for example, in Hopcroft and Ullman (1979:325). It is easy to show that the more restricted version entails no loss of generality; again see Hopcroft and Ullman (1979:328–330). Our example illustrates a particularly restricted version of satisfiability where there are exactly three so-called literals per clause, dubbed *3SAT*. As we shall see in chapter 2, this restricted problem is just as hard as the unrestricted version of satisfiability, where there are any number of literals per clause.

	Problem size, n		
Time complexity	**10**	**50**	**100**
n^3	.001 second	.125 second	1.0 second
2^n	.001 second	35.7 years	10^{15} centuries

Figure 1.1: Exponential growth limits solvable problem sizes. A cubic-time algorithm (second line in the table) can solve problems of size 100 in a second, while a corresponding exponential time algorithm (last line) would take far too long. The entries in the table, modeled after Garey and Johnson (1979), assume that each algorithm instruction takes 1 microsecond, but the shape of the curve relating problem size to processing time is more important than the exact time values.

There's good reason to identify SAT as a prototypical computationally *intractable* problem. Let us see why. If you try to solve this example in your head, you'll quickly note that you mentally run through *every* possible combination of assignments, testing each in turn. With n binary-valued variables in an arbitrary formula, there are 2^n possible truth-value assignments to test. In fact, every known algorithm for solving this problem takes at least time proportional to 2^n, or *exponential time*, where the number of variables n can obviously rise proportionally with length of the tested input formula.

Figure 1.1, adapted from Garey and Johnson (1979), shows why we say that P corresponds to the class of computationally tractable problems, while problems for which only exponential solution algorithms are known—including SAT—are intractable. Assuming that a solution algorithm's running time is proportional to the problem size to be solved, the first line in the table shows that if an algorithm takes time proportional to n^3, then even large-sized problems can be done in a second or less. But we can't wait around for an exponential-time algorithm working on a problem of the same size.

Of course, there are familiar pitfalls in comparing exponential time and polynomial time algorithms—n^{10000} can be quite slow, particularly for smaller values of n, when compared to 2^n or $2^{0.01n}$. But in fact it turns out that this bifurcation fares quite well in classifying naturally occurring com-

puter science problems; if a problem is efficiently solvable at all, it will in general be solvable by a polynomial algorithm of low degree, and this seems to hold for linguistically relevant problems as well.[5]

What class of problems does SAT fall into, then? The difficult part about SAT seems to be guessing all the possible truth assignments—2^n of them, for n distinct variables. Suppose we had a computer that could try out all these possible combinations, in parallel, without getting "charged" for this extra ability. We might imagine such a computer to have a "guessing" component (a factory-added option) that writes down a guess—just a list—of the *true* and *false* assignments. Given any SAT formula, we could verify quite quickly whether any guess works: just scan the formula, checking the tentative assignment along the way. It should be clear that checking a guess will not take very long, proportional to the length of the tested formula (we will have to scan down our guess list a few times, but nothing worse than that; since the list is proportional to n in length, to be conservative we could say that we will have to scan it n times, for a total time proportional to n^2). In short, *checking* or *verifying* one guess will take no more than polynomial time and so is in $\mathcal{P}$, and tractable.

Therefore, our hypothetical computer that can try out *all* guesses in parallel, without being charged for guessing wrong, would be able to solve SAT in polynomial time. Such a computer is called *nondeterministic* (for a more precise definition, see chapter 2, section 2.1), and the class of problems solvable by a $\mathcal{N}$ondeterministic computer in $\mathcal{P}$olynomial time is dubbed $\mathcal{NP}$.

1.2.3 Problems with no efficient solution algorithms

Plainly, all the problems in $\mathcal{P}$ are also in $\mathcal{NP}$, because a problem solvable in deterministic polynomial time can be solved by our guessing computer simply by "switching off" the guessing feature. But SAT is in $\mathcal{NP}$ and not known to be in $\mathcal{P}$. For the practically minded, this poses a problem, because our hypothetical guessing computer doesn't really exist; all we have are deterministic computers, fast or slow, and with the best algorithms we know these all take exponential time to solve general SAT instances. (See section 1.4 for a discussion of the potentials for parallel computation.) In fact, complexity

[5]However, there are some linguistic formalisms whose language recognition problems take time proportional to n^6, such as Head Grammars (Pollard 1984), and some linguistic problems such as morphological analysis tend to have short inputs. We take up these matters again in chapter 2 and elsewhere.

theorists have discovered many hundreds of problems like SAT, for which only exponential-time deterministic algorithms are known, but which have efficient nondeterministic solutions. For this reason, among others, computer scientists strongly suspect that $P \neq NP$.

Complexity theory says more than this, however: it tells us that problems like SAT serve to "summarize" the complexity of an entire class like NP, in the sense that *if* we had an algorithm for solving SAT in deterministic polynomial time then we would have an algorithm for solving *all* the problems in NP in deterministic polynomial time, and we would have $P = NP$. (We'll see why that's so just below and in the next section.) Such problems are dubbed *NP-hard*, since they are "as hard as" any problem in NP. If an NP-hard problem is also known to be *in* NP—solvable by our hypothetical guessing computer, as we showed SAT to be—then we say that it is *NP-complete*.

Roughly speaking then, all NP-complete problems like SAT are in the same computational boat: solvable, so far as we know, only by exponential-time algorithms. Because there are many hundreds of such problems, because none seems to be tractable, and because the tractability of any one of them would imply the tractability of all, the $P \neq NP$ hypothesis is correspondingly strengthened. In short, showing that a problem is NP-hard or NP-complete is enough to show that it's unlikely to be efficiently solvable by computer. We stress once more that such a result about a *problem's* complexity holds independently of any algorithm's complexity and independently of any ordinary computer model.[6]

We pause here to clear up one technical point. Frequently we will contrast polynomial-time algorithms with combinatorial search and other exponential-time algorithms. However, even if $P \neq NP$—as seems overwhelmingly likely—it might turn out that the true complexity of hard problems in NP lies somewhere between polynomial time and exponential time. For instance, the function $n^{\log n}$ outstrips any polynomial because (informally) its degree keeps slowly increasing, but the function grows less rapidly than an exponential function (Hopcroft and Ullman 1979:341). However, because only exponential-time algorithms are currently known for NP-complete problems, we will continue to say informally that problems in NP seem to require combinatorial search.

[6] We discuss familiar caveats to this claim in chapter 2; these include the possibility of heuristics that work for problems encountered in practice, the effect of preprocessing, and the possibility of parallel speedup.

1.2.4 The method of reduction

Because demonstrating that a problem is NP-hard or NP-complete forms the linchpin for the results described in the rest of the book, we will briefly describe the key idea behind this method and, in the next section, illustrate how to apply it to a very simple, artificial grammatical system; for a more formal, systematic discussion, see chapter 2.

Showing that one problem is computationally as difficult as another relies on the technique of *problem transformation* or *reduction*, illustrated in figure 1.2. Given a new problem T, there are three steps to demonstrating that T is NP-hard, and there's a fourth to show T is NP-complete:

1. Start with some known NP-hard (or NP-complete) problem S. Selection of S is usually based on some plain correspondence between S and T (see the example just below and chapter 2 for further examples).
2. Construct a mapping Π (called a *reduction*) from instances of the known problem S to instances of the new problem T, and show that the mapping takes polynomial time or less to compute. In this book, problems will always be posed as *decision problems* that have either Yes or No answers, *e.g.*, is a particular Boolean formula satisfiable or not?[7]
3. Show that Π preserves Yes and No answers to problems. That is, if S has a Yes answer on some instance x, then T must have a Yes answer on its instance $\Pi(x)$, and similarly for No answers.
4. If an NP-completeness proof is desired, show in addition that T is in $\mathcal{NP}$, that is, can be solved by a "guessing" computer in polynomial time. Note that this step isn't required to demonstrate computational intractability, because an NP-hard problem is at least as hard as any problem in $\mathcal{NP}$.

If one likes to think in terms of subroutines, then such a *polynomial-time reduction* shows that the new problem T must be at least as hard to solve as the problem S of known complexity, for the following reason. If we had a polynomial-time subroutine for solving T, then S could also be solved in polynomial time. We could use the mapping Π to convert instances of S into instances of T, and then use the polynomial-time subroutine for solving

[7] Well-defined problems that don't have simple Yes/No answers—such as "what's the shortest cycle in this graph?"—can always be reformulated as decision problems; see Garey and Johnson 1979:19–21.

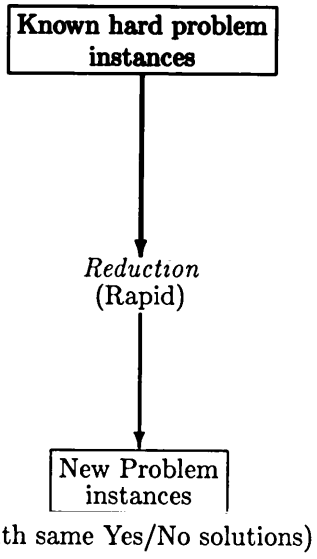

Figure 1.2: Reduction shows that a new problem is complex by rapidly transforming instances of a known difficult problem to a new problem, with the same Yes/No answers.

T on this converted problem. The answer returned for T always coincides with the original answer for S, because Π is known to preserve answers. Because we also know that Π can be computed in polynomial time, and since the composition of two polynomial-time subroutines is also polynomial-time, this procedure would solve S in polynomial time. But the problem S, such as SAT, is NP-hard and not thought to be solvable in polynomial time. Therefore either S and all other problems in $\mathcal{NP}$ are efficiently solvable, a tremendous surprise, or else no polynomial-time subroutine for T exists.

In short, our reduction proves that the new problem T is at least as hard the old one S *with respect to polynomial time reductions.* Either T is even harder than S, or else the two are in the same computational boat. (One can now see why the problem transformation itself must be "fast"—polynomial time or better—for otherwise we would introduce spurious complexity and could not make this argument.)

Before proceeding with a more linguistically oriented example in the next section, we'll consider the obvious question of how all this can ever get

started. Step 1 of the reduction technique demands that we start with a known NP-hard or NP-complete problem, and we've said several times that SAT fits the bill. But how does one get things off the ground to show that SAT is NP-complete? There is no choice but to confront the definition of NP-hardness directly: we must show that, given any algorithm that runs on our hypothetical "guessing" computer in polynomial time, we can (in polynomial time) build a corresponding SAT problem that gives the same answers as that algorithm. Such a construction shows that SAT instances can "simulate" any polynomial-time nondeterministic algorithm on any ordinary computer, and so SAT is NP-hard. In fact, SAT must also be NP-complete, as it's clearly solvable by our guessing computer.[8] Starting with SAT as a base, we can begin to use reduction to show that other problems are NP-hard or NP-complete. Section 2.2 in the next chapter shows how this is done, including how to transform SAT to 3SAT.

1.3 A Simple Grammatical Reduction

To give an introduction to how we use reduction to analyze grammatical formalisms, in this section we consider a very simple and artificial grammatical example. Readers familiar with how reductions work may skip this discussion; chapter 3 contains a more formal treatment of a similar problem.

Our grammatical system expresses two basic linguistic processes: lexical ambiguity (words can be either nouns or verbs) and agreement (as in subject-verb agreement in English). These processes surface in many natural languages in other guises, for example, languages with case agreement between nouns and verbs.

In particular, our artificial grammatical system exhibits a special kind of global agreement: once a particular word is picked as a noun or a verb in a sentence, any later use of that word in the same sentence must agree with the previous one—and so its syntactic category must also be the same. (One might like to think of this as a sort of syntactic analog of the vowel harmony that appears within words in languages like Turkish: all the vowels of a series of Turkish suffixes may have to agree in certain features with a preceding root vowel.)

[8] Chapter 2 gives more detail on this. Garey and Johnson (1979:38–44) give a full proof, originally by Cook (1971).

The one exception to this agreement is when a word ends in a suffix *s*. Then, it must *disagree* with the same preceding or following word without the suffix. Finally, this language's sentences contain any number of clauses, with three words per clause, and each clause must contain at least one verb.

For example, if we temporarily ignored (for brevity) the requirement that a clause must contain three words, then

apple bananas, apples banana, AND apples bananas

could be a sentence. It's hard to tell what's a noun and what's a verb, given the lexical ambiguity that holds; if *apple* is a verb, then *apples* must not be, so *banana* is the only possible verb in the second clause—so far, so good. But then *apples* and *bananas* must both be nouns, and the last clause has no verb. Consequently, *apple* has to be a noun instead, and *bananas* must be the verb of the first clause. *Banana* is then a noun, but we already know *apples* is a verb, so the second clause is okay. Finally, the last clause now has two verbs, so the whole thing is a sentence (except for the three-word requirement).

How hard will it be to recognize sentences generated by a grammatical system like this? One might try many different algorithms, and never be sure of having found the best one. But it is precisely here that complexity theory's power comes to the fore. A simple reduction can tell us that this general problem is computationally intractable—NP-hard—and almost certainly, there's no easy way to recognize the sentences of languages like this.

It should be clear that this artificial grammatical system is but a thinly disguised version of the restricted SAT problem—known as 3SAT—where there are exactly three literals (negated or unnegated variables) per clause. Some proofs are simplified if 3SAT is defined to require exactly three *distinct* literals per clause, though we will not always impose this requirement.[9]

Given any 3SAT instance, it is easy to quickly transform it into a language recognition problem in our grammatical framework, with corresponding Yes/No answers. The verb-noun ambiguity stands for whether a literal gets assigned **true** or **false**; agreement together with disagreement via the *s* marker replaces truth assignment consistency, so that if an x is assigned **true** (that is, is a verb) in one place, it has the same value everywhere, and if it is $\overline{x}$ (has the *s* marker) it gets the opposite value; finally, demanding one verb per

[9]It is easy to show that 3SAT—like SAT—is NP-complete; see section 2.2. Also, it is easy to show that the restriction to distinct literals is inessential.

clause is just like requiring one true literal per satisfiability clause. The actual transformation simply replaces variable names with words, adds *s* markers to words corresponding to negated literals, tidies things up by setting off each clause with a comma, and deletes the extraneous logical notation. The result is a sentence to test for membership in the language generated by our artificial grammar. Plainly, this conversion can be done in polynomial time, so we've satisfied steps 1 and 2 of our reduction technique.[10]

Figure 1.3 shows the reduction procedure in action on one example problem instance. The figure shows what happens to the Boolean formula given earlier:

$$(x \vee \overline{y} \vee \overline{z}) \wedge (y \vee z \vee u) \wedge (x \vee z \vee \overline{u}) \wedge (\overline{x} \vee y \vee u)$$

We can convert this satisfiability formula to a possible sentence in our hypothetical language by turning u, x, y, and z into words (*e.g.*, *apple*, *banana*, *carrot*, ...), adding the disagreement marker *s* when required, putting a comma after each clause (as you might do in English), and sticking an *and* before the last clause. Running this through our reduction processor yields a sentence with four clauses of three words each:

apple bananas carrots, banana carrot dandelion, apple carrot dandelions, AND apples banana dandelion

We now check step 3 of the reduction technique: answer preservation. The output sentence is grammatical in our artificial system if and only if each clause contains at least one verb. But this is so if and only if the original formula was satisfiable. Since this holds no matter what formula we started with, the transformation preserves problem solutions, as desired. We conclude that the new grammatical formalism can pose problems that are NP-hard. Remember how potent this result is: we now know that *no matter what* algorithm or ordinary computer we pick, this grammatical *problem* is computationally intractable.

Our example also illustrates a few subtle points about problem reductions to keep in mind throughout the remainder of the book. When a reduction involves constructing some grammar G, the language $L(G)$ that the grammar generates will often be a particularly simple language; for instance,

[10] We can just sweep through the original formula left-to-right; the only thing to keep track of is which variables (words) we've already seen, and this we can do by writing these down in a list we (at worst) have to rescan n times.

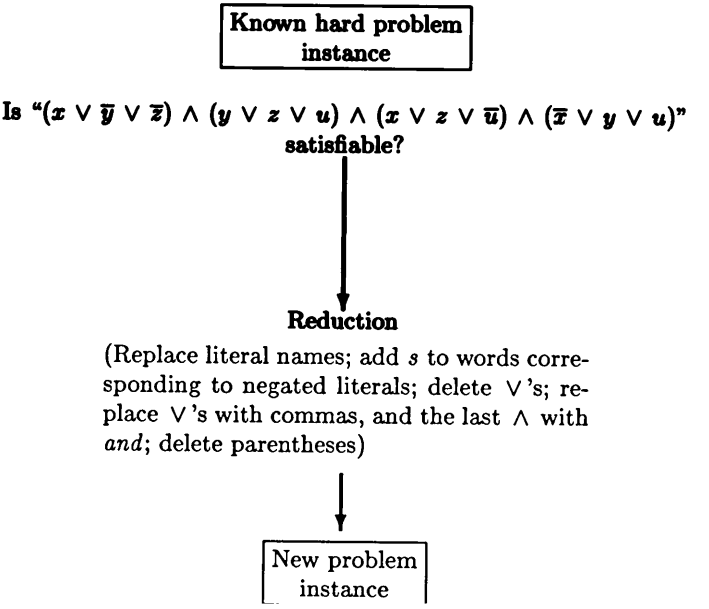

Figure 1.3: A reduction from 3SAT shows ambiguity-plus-agreement to be hard. This example shows how just one 3SAT problem instance may be rapidly transformed to a corresponding sentence to test for membership in an artificial grammar. In this case, the original formula is satisfiable with x, y, and z set to **true**, and the corresponding sentence is grammatical, so Yes answers to the original and new problems coincide as desired.

$L(G)$ might contain only the single string "#", or $L(G)$ might be the empty set. (Section 5.7.2 uses an example of this sort.) It's important to distinguish between the complexity of the set $L(G)$ (certainly trivial, if $L(G) = \{\#\}$) and the difficulty of figuring out *from the grammar* G whether $L(G)$ contains some string. For example, we might know that no matter what happens, the reduction *always* constructs a grammar that generates either the empty set or the set $\{\#\}$—either way, a language of trivial complexity—yet it might still be very hard to figure out *which one* of those two possible languages a given G would generate. In technical terms, this means we must distinguish the complexity of the *recognition problem for some class of grammars* from

the complexity of *an individual language that some grammar from the class generates.*

A second, related point is the distinction between the *input* to the problem transformation algorithm (an instance of a problem of known complexity) and the *string inputs* to the problems of known and unknown complexity; these problem inputs are typically simple strings. In all, then, there are three distinct "inputs" to keep track of, and these can be easily confused when all three are string languages that look alike.

To summarize, while our example is artificial, our method and moral are not. Chapters 3–9 use exactly the same technique. The only difference is that later on we'll work with real grammatical formalisms, use fancier reductions, and sometimes use other hard problems besides SAT. (Section 2.2 outlines these alternative problems.)

1.4 The Idealizations of Complexity Theory

Having seen a bit of what complexity theory is about, and how we can use it to show that a grammatical formalism can pose intractable (NP-hard) problems, we now step back a bit and question whether this technique—like all mathematical tools—commits us to idealizations that lead us in the right direction. We believe the answer is Yes, and in this section we'll briefly survey why we think so. In the next chapter, sections 2.3 and 2.4 delve more deeply into each of these issues (and consider some others besides).

To evaluate the idealizations of complexity theory, we must reconsider our goals in using it. Complexity theory can tell us *why* the processing problems for a formalized grammatical system have the complexity they do, whether the problems are easy or difficult. By probing sources of processing difficulty, it can suggest ways in which the formalism and processing methods may fail to reflect the special structure of a problem. Thus, complexity theory can tell us where to look for new constraints on an overly powerful system, whether they are imposed as constraints on the grammatical formalism or as performance constraints. It can also help isolate unnatural restrictions on suspiciously simple systems. In a nutshell, these goals require that our idealizations must be *natural* ones—in the sense that they don't run roughshod over the grammatical systems themselves, contorting them so that we lose touch with what we want to discover.

We feel that the potential "unnaturalness" surrounding mathematical results in general must be addressed: are the grammatical problems posed in such a way that they lead to the insights we desire? Although a discussion of those insights must wait for later chapters, here we can at least show that the idealizations we've adopted are designed to be as natural and nonartificial as possible. Some of our basic idealizations seem essential: given current ignorance about human brainpower, we want to adopt an approach as independent of algorithm and machines as possible, and that's exactly what the theory buys us. Other idealizations need more careful support because they seem more artificial. The following sections will address several issues. First, there are questions about complexity theory's measures of problem complexity; we'll consider the assumption that problems can grow without bound, the relevance to grammatical investigations of linguistically bizarre NP-complete problems such as SAT, and the status of the more traditional "complexity" yardstick of weak generative capacity. Next, we'll discuss our assumption that we should study the complexity of grammatical *systems*, which corresponds to posing certain kinds of problems (universal problems) rather than others; and finally, we'll turn to our reliance on invariance with respect to *serial* computer models.

1.4.1 The role of arbitrarily large problems

Complexity theory assumes that problems can grow arbitrarily in size; for instance, the length of Boolean formulas in the SAT problem can grow arbitrarily, and algorithms for solving SAT must work on an infinity of SAT instances. We adopt this idealization wholeheartedly simply because we have to in order to use complexity theory at all. That's because the complexity of a problem that's bounded in size is actually *zero*, according to the way complexity theory works—so the theory would tell us nothing at all if we assumed that grammatical problems couldn't grow without bound.[11]

Some might question this infinity-based assumption for natural language. After all, the sentences we encounter are all of bounded length—

[11]That result is not as strange as it first appears. It's simply because one can solve, in advance, *all* the problems less than a certain size, and store the results in a giant table. Then the small problems may be rapidly retrieved, in bounded time, and large problems can be rejected out of hand as soon as we've seen enough symbols to realize they're too big. In this case, then, complexity doesn't depend on the problem size at all. For instance, we can certainly number and then solve all the satisfiability problems less than 8 clauses long with 3 literals per clause.

certainly less than 100 words long. The number of distinct words in a natural language, though very large, is also bounded. Therefore, natural language problems are always bounded in size; they can't grow as complexity theory assumes. Aren't then the complexity results irrelevant because they apply only to problems with arbitrarily long sentences or arbitrarily large dictionaries, while natural languages all deal with finite-sized problems?

It is comforting to see that this argument explodes on complexity theoretic grounds just as it does in introductory linguistics classes. The familiar linguistic refrain against finiteness runs like this: Classifying a language as finite or not isn't our *raison d'être*. The question appears in a different light if our goal is to determine the form and content of linguistic knowledge. When we say that languages are infinite, we don't really intend a simple classification. Instead, what we mean is that once we have identified the principles that seem to govern the construction of sentences of reasonable length, there doesn't seem to be any natural bound on the operation of those principles. The principles—that is, the principles of grammar—characterize indefinitely long sentences, but very long sentences aren't used in practice because of other factors that don't seem to have anything to do with how sentences are put together. If humans had more memory, greater lung capacity, and longer lifespan—so the standard response goes—then the apparent bound on the length of sentences would be removed.

In just the same way, complexity theorists standardly generalize problems along natural dimensions: for instance, they study the playing of checkers on an arbitrary $n \times n$ board, rather than "real" checkers, because then they can use complexity theory to study the structure and difficulty of the problem. The problem with looking at problems of bounded size is that results are distorted by the boring possibility of just writing down all the answers beforehand. If we study checkers as a bounded game, it comes out (counterintuitively!) as having no appreciable complexity—just calculate all the moves in advance—but if we study arbitrary $n \times n$ boards, we learn that checkers is computationally intractable (as we suspected).[12] Thus, the idealization of unboundedness is necessary for the same reason in both linguistics and complexity theory: by studying problems of arbitrary size we remove factors that would obscure the structure of the domain we're studying.

[12]In fact, this checkers generalization is probably harder than problems in $\mathcal{NP}$; it is PSPACE-hard. See Garey and Johnson (1979:173) for this result and chapter 2 for a definition of *PSPACE*, consisting of the problems that can be solved by an ordinary computer in polynomial *space*.

Related is the question of whether it's valid to place a bound on some particular parameter k of a problem—such as the length of a grammar rule or the number of variables in a SAT problem—in order to remove a factor from the complexity formula, while leaving other parameters of the problem unbounded. Here, the answer depends on the details of the problem. As a general rule, we obscure complexity instead of improving it if we simply impose a bound. For instance, if the complexity of our algorithm is $2^k \cdot n^3$, we haven't helped anything if we set a bound of $k = 50$ and then bound our computational effort by $K \cdot n^3$ where $K = 2^{50}$. But this kind of truncation can be genuinely valid if a linguistically justified bound produces a small constant in the complexity formula, or (more interestingly) if the bound can actually be exploited in an algorithm—for instance by using resolution on 2SAT (see section 1.4.2) or by building some small and clever table into the program.[13] (Sections 7.10 and 9.1.2 discuss computational and linguistic considerations that bear on the possibility of limiting the length of one computationally troublesome kind of grammar rule.)

Except in these special situations, truncation buys nothing but obfuscation, for the algorithm will behave just the same on the truncated problem as it does on the full problem—except that its complexity curve will artificially level out when the bound is reached. For instance, if we use a standard exponential algorithm to process SAT formulas, but limit the formulas to at most 10 distinct variables, we can expect the complexity curve to resemble the one shown in figure 1.4. Before the bound on variables is reached, longer formulas can get exponentially harder because they can contain more and more variables whose truth-values must be guessed; but after the bound is reached, runtime will increase at a much slower rate.

Since complexity theory deals in asymptotes, the complexity formula will be derived from the flattened-out portion of the curve, and the problem will look easy. But the initial, exponentially growing portion of the curve tells a different tale—naturally so, since by hypothesis we're using the same exponential algorithm as always. Nothing about any special structure of the

[13] In addition, more sophisticated "truncation" moves are possible. S. Weinstein has suggested that one option for a theory of performance involves quickly transforming a competence grammar G into a performance grammar $f(G)$ that can be rapidly processed. The function f "truncates" the full grammar in such a way that the symmetric difference between the languages $L(G)$ and $L(f(G))$ is negligible, in some natural sense that remains to be clarified; for instance, the truncated grammar might reject center-embedding or flatten deeply right-branching constructions. Many questions arise, among them the status of G and the relationship between the formalism(s) in which G and $f(G)$ are expressed.

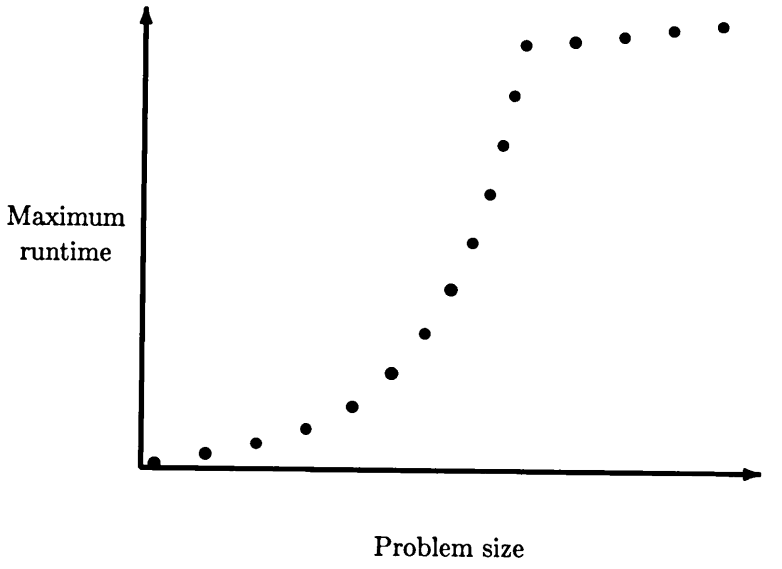

Figure 1.4: If a bounded problem is derived by pure truncation of a difficult parameter, with no change in an underlying exponential algorithm, we can expect runtime to grow exponentially at first and then level off when the artificial bound is reached.

truncated problem has been exploited; there's only the happy circumstance of a finite search space, and lurking below a patina of efficiency, brute-force search still reigns. Clearly, the initial region of the curve tells us more about the structure of the algorithm than the artificially flattened part. In a case like this, truncation is not an appropriate move because it only muddies the water. Just as in the linguistic case—when the structure of grammatical constraints could be better understood by considering unbounded problems—the structure of the algorithm is better revealed by considering how it operates on unlimited cases, without the truncation bound.

We conclude that if we want to use computational complexity theory then it makes sense to think of natural language processing problems *as if* they are of arbitrary size, even if they are not. To do otherwise is to risk masking the symptoms of exponential-time search through artificial means.

1.4.2 Why hard problems needn't be artificial

A second basic assumption of our approach is that the $\mathcal{P}$–$\mathcal{NP}$ distinction isn't just an artificial one for natural languages: that hard problems like SAT do turn up in natural grammatical systems, and what's more, such problems do highlight the information processing structure of natural grammars.

The worry about artificiality seems to boil down to this: problems like SAT don't seem to be much like anything that any natural language processor would ever compute. Indeed, if by hypothesis natural problems are easier than SAT, then we might automatically avoid computational difficulty by using the frameworks only for real linguistic tasks instead of mathematical troublemaking.

Again, both our natural language analyses and complexity theory itself dismiss such worries as groundless. First, natural grammars *do* contain hard problems: as chapter 3 shows, the difficulty of processing sentences like BUFFALO BUFFALO BUFFALO seems to arise precisely because grammars can pose difficult problems. Similarly, chapter 5's spelling-change and dictionary system is computationally intractable as shown by a reduction that at least superficially mirrors ordinary language processes like vowel harmony. Finally, chapter 8 and appendix B show that generalized phrase structure grammar parsing can be difficult in practice.

Restrictions to "natural" cases, then, won't automatically save us from intractability. But this is no surprise to the complexity theorist. Here too, examples demonstrate that unless one exploits the special information structure of a problem, "natural" restrictions may not suffice to win processing efficiency.

A good example is a restricted version of SAT where there are two literals per clause, known as 2SAT. 2SAT is easier than 3SAT—it's in $\mathcal{P}$—and so doesn't require exponential time for solution; yet if you take the usual exponential algorithm for SAT and expect it to run faster on 2SAT problems because they're easier, you will be sorely disappointed. The SAT algorithm will simply do the same kind of combinatorial search as before and will take exponential time. One must use a specialized algorithm such as resolution theorem proving to get any mileage out of the special structure of this restricted problem.[14] There's no reason why the same thing shouldn't happen

[14]In particular, the "special" structure is that there are two literals per clause. When resolution combines two such clauses together, the resolvent, by definition, is no longer

with grammatical machinery—a problem that's not intrinsically hard can be made difficult through failings of the grammatical framework, perhaps not obvious ones. In fact, section 7.8.1 gives an example of an easy problem that's made to look difficult when it's encoded in a context-free grammar.

1.4.3 Weak generative capacity can be misleading

Like our complexity tools, considerations of weak generative capacity can aid us in linguistic investigations; recall Chomsky's (1956) early demonstration of the inadequacy of finite-state descriptions of natural languages, which was based partially on grounds of weak generative capacity. Yet for many reasons, weak generative capacity alone may not give good clues about the appropriateness or processing difficulty of a grammatical formalism—one fundamental reason that we generally reject weak generative capacity analysis as too blunt and focus on complexity classifications instead.

A weak-generative-capacity restriction to strictly context-free languages is often thought to guarantee efficient parsability, but no such result holds. The reason, briefly, is that some context-free languages are generated only by very large context-free grammars—and grammar size *does* affect parsing time for all known general context-free parsing algorithms. We won't belabor this point here, as it is adequately discussed in chapters 7 and 8.

Similarly, models based on finite-state automata are often considered the hallmark of computational efficiency. Yet they, too, can lead one astray. While it is true that some finite-state problems are easy, other finite-state problems can be computationally costly. One must carefully examine *how* finite-state machinery is being used before pronouncing it safe from computational intractability; oversights have led to much confusion in the linguistics literature. Most researchers know casually that it's fast to figure out whether a sentence can be accepted or rejected by a finite-state automaton. No search is involved; the machine just processes the sentence one word at a time, and at the end, it just gives a Yes or No answer—the sentence either is or is not accepted. In short, the problem of *finite-state recognition* is easy.

But one cannot always rely on this approach to model all finite-state processes. For example, suppose we wanted to know the complexity of finite-

than the length of either of the original clauses. This monotonicity allows resolution to work in polynomial time. If one tries the same trick with 3SAT, then one quickly discovers that resolved clauses can grow in length, frustrating a polynomial time solution.

state *parsing*. That is, suppose we wanted not simply a Yes/No nod from our automaton, but a detailed description of the sentence's internal structure—perhaps a sequence of word category names. After all, this cuts closer to the heart of what we want from natural language analysis. But it looks like a harder problem, because it demands more information. Do our previous results about mere finite-state recognition apply? (In general, parsing is harder than recognition because a parsing algorithm must output a representation of how a sentence was derived with respect to a particular grammar, not merely a Yes/No recognition answer.)

Even if a problem is carefully posed, a solution in terms of finite-state machinery may be inappropriate if it does not accurately reflect the underlying constraints of a language. Rather, the finite-state character may be an accidental by-product, one that has little to do with the nature of the constraints that characterize the problem. In such a case, considerations of weak generative capacity are uninformative at best and misleading at worst. As was noted many years ago, weak generative capacity analysis serves as a kind of "stress test" that doesn't tell us much unless a grammar *fails* the test:

> The study of weak generative capacity is of rather marginal linguistic interest. It is important only in those cases where some proposed theory fails even in weak generative capacity—that is, where there is some natural language even the *sentences* of which cannot be enumerated by any grammar permitted by this theory .... It is important to note, however, that the fundamental defect of [many systems] is not their limitation in weak generative capacity but rather their many inadequacies in strong generative capacity .... Presumably, discussion of weak generative capacity marks only a very early and primitive stage of the study of generative grammar. (Chomsky 1965:60f)

Flaws in a formal system can easily go undetected by weak generative capacity analysis.

To see what goes wrong in a specific example, consider another simple artificial language, a *bounded palindrome language*—a set of sentences shorter than some fixed length k that can be read the same backwards or forwards. Over the alphabet a, b, c with a length restriction of 3, this gives us the language a, b, c, aa, bb, cc, aaa, aba, aca, bab, bbb, bcb, cac, cbc, ccc. Now, it is well known that an infinite palindrome language over the same alphabet cannot be generated by any finite-state grammar; the implicit mirror-

image pairing between similar letters demands a context-free system. But our k-bounded palindrome language contains only a finite number of sentences, hence is technically and mechanically finite-state; therefore, the finite-state framework fails to break under the stress test of generative capacity.

But despite the fact that the language is finite-state, it is seriously misleading to stop and conclude that the finite-state framework accurately expresses the underlying constraints of this language. Just as with our earlier 2SAT vs. 3SAT example, it's instructive to consider the details of what's happening. *What kind* of finite-state machine generates our bounded palindrome language? Going through the tedious exercise of constructing the machine, say for $k = 6$, one finds that the underlying automaton, though indeed finite-state, represents a kind of huge brute-force encoding of *all* possible sentences—just a list, if you will. And just as with our exhaustive combinatorial algorithms, *nothing* about the special mirror-image structure of palindromes is exploited; such a machine could have just as easily encoded a random, finite list of sentences. It makes sense to remove this unilluminating accident by idealizing to an infinite palindrome language—which isn't finite-state—and then imposing boundedness as a separate condition.

Many examples of this kind also exist in natural languages. For example, many *reduplicative* processes—the kind that double constituents like syllables, roots, affixes, and so forth—in fact duplicate only a bounded amount of material. Technically, then, they can be encoded with context-free or even finite-state machinery, though the related language $\{ww\}$ where w ranges over unbounded strings is strictly not context-free. But clearly, the reduplicated material's boundedness may tell us nothing about the true nature of the constraints that are involved. In this case too, the machinery may pass the weak generative capacity test for accidental reasons. The point is that simple classification—the question of whether natural languages are context-free, for instance—doesn't have a privileged position in linguistic investigations. Unless very carefully used, the classification scheme of weak generative capacity may well be too blunt to tell us anything illuminating about natural languages.[15] We prefer complexity theory because it gives us more direct insight into the structure of grammatical problems.

[15]Rounds, Manaster-Ramer, and Friedman (1986) have more to say on related points.

1.4.4 Universal vs. fixed-language recognition problems

Beyond these basic idealizations, we have posed the grammatical problems described in the rest of this book in a particular way. Because our problem descriptions sometimes seem at odds with those familiar from the tradition of weak generative capacity analysis, we shall briefly review why we think our approach heads in the right direction; in the next chapter, section 2.3 provides a more complete discussion of the same issue.

We aim to study the complexity of entire *families* of grammars—namely, those specified by some linguistic formalism, like lexical-functional grammar or generalized phrase structure grammar. This leads most naturally to the following complexity problems, which are most often called *universal problems* because they deal with an entire grammatical class:

> Given a grammar G (in some grammatical framework) and a string x, is x in the language generated by G?

We contrast this way of posing complexity problems with the way such problems are often stated in the weak generative capacity tradition, dubbed *fixed language recognition problems* (FLR problems):

> Given a string x, is x in some independently specified set of strings L?

These two problems may look very much alike, but they are not. Universal grammar problems contain two variables: the grammar and an input sentence. Fixed language recognition problems contain one variable, just the sentence to recognize. No particular grammar is specified—just as no particular grammar is mentioned when we say that a certain string language is or isn't context-free, in the weak generative capacity approach. Generally speaking, universal problems are harder, because the grammar is variable: a potential solution algorithm must be braced for any possible grammar thrown at it. In contrast, FLR problems are easier: because one is permitted to vary the grammar at will to get the most efficient algorithm possible, and because no grammar is mentioned in the problem, there's no "grammar size" parameter to appear in complexity formulas.[16]

[16]For example, the FLR problem for context-free *languages* takes only time proportional to n^3, as is well known (Hopcroft and Ullman 1979). However, the corresponding universal problem, where the grammar must be taken directly into account, is much harder: it is

Even though FLR problems are usually easier in a formal sense, they are misleadingly so. In a nutshell, FLR problems ignore grammars, parsing, and complexity theory practice, while universal problems focus on all these things in the right way—they explicitly grapple with grammars instead of languages, take into account parsing difficulties, and accord with complexity theory practice:

- Universal problems study entire grammatical families by definition, while FLR problems consider only language complexity and so allow one to vary the grammar at will. Implicitly, an FLR problem can allow one to completely ignore the grammatical formalism under study just to get the simplest language complexity possible. But this cuts directly against our aim to study properties of the grammatical formalisms themselves, not just the languages they happen to generate. In addition, if one believes that grammars, not languages, are mentally represented, acquired, and used, then the universal problem is more appropriate.
- Universal problems consider all relevant inputs to parsing problems, while FLR problems do not. First of all, we're interested in parsing with respect to linguistically relevant grammars; we're not just interested in language recognition problems. Second, we know that grammar size frequently enters into the running time of parsing algorithms, usually multiplied by sentence length. For example, the maximum time to recognize a sentence of length n of a general context-free language using the Earley algorithm is proportional to $|G|^2 \cdot n^3$ where $|G|$ is the size of the grammar, measured as the total number of of symbols it takes to write the grammar down (Earley 1968). What's more, it's typically the grammar size that dominates: because a natural language grammar will have several hundred rules but a sentence will be just a dozen words long, it's often been noted that grammar size contributes more than the input sentence length to parsing time. (See Berwick and Weinberg (1984), as well as appendix B for some evidence of this effect in generalized phrase structure grammars.) Because this is a relevant input to the final complexity tally, we should explicitly consider it.
- A survey of the computational literature confirms that universal problems are widely adopted, for many of the reasons sketched above. For

P-complete (as difficult as any problem that takes deterministic time n^j) (Jones and Laaser 1976).

> example, Hopcroft and Ullman (1979:139) define the context-free grammar recognition problem as follows: "Given a context-free grammar G and a string x ... is x in [the language generated by G]?" Garey and Johnson (1979), in a standard reference work in the field of computational complexity, give all 10 automata and language recognition problems covered in the book (1979:265–271) in universal form: "Given an instance of a machine/grammar and an input, does the machine/grammar accept the input?"

All of these considerations favor the use of universal problems, but it is also fair to ask whether one could somehow preprocess a problem in some way—particularly a problem that includes a grammar—to bypass apparent computational intractability. After all, a child learning language may have a lot of time at its disposal to discover some compact, highly efficient grammatical form to use. Similarly, people are thought to use just *one* grammar to process sentences, not a family of grammars. So isn't the FLR model the right one after all?

The preprocessing issue—essentially, the issue of compilation—is a subtle one that we'll address in detail in the next chapter (section 2.3). However, we can summarize our main points here. Compilation suffers from a number of defects.

First of all, compilation is neither computationally free nor even always computationally possible. Compilation cannot be invoked simply as a promissory note; one must at least hint at an effective compilation step.

Second, if we permit just *any* sort of preprocessing changes to the grammar in order to get a language that is easy to process, then there is a tremendous temptation to ignore the grammatical formalism and allow clever programming (the unspecified preprocessing) to take over. If, on the other hand, we believe that grammars are incorporated rather directly into models of language use, then this independence seems too high a price to pay.

Finally, *known* compilation steps for spelling change and dictionary retrieval systems, lexical-functional grammar, generalized phrase structure grammars, and subsystems of GPSGs known as ID/LP grammars all fail: they cannot rescue us from computational intractability. Typically, what happens is that compilation expands the grammar size so much that parsing

algorithms take exponential time.[17] See chapters 4, 5, 7, and 8 for the details, and chapter 2 (section 2.3) for a more thorough discussion of the compilation issue.

1.4.5 The effect of parallel computation

A final issue is that, for the most part, the complexity classes we use here remain firmly wedded to what we've been calling "ordinary" computers—serial computers that execute one instruction at a time. We have already stressed that complexity results are invariant with respect to a wide range of such sequential computer models.

This invariance is a plus—if the sequential computer model is the right kind of idealization. However, since many believe the brain uses some sort of parallel computation, it is important to ask whether a shift to parallel computers would make any difference for our complexity probes. Complexity researchers have developed a set of general models for describing parallel computation that subsume all parallel machines either proposed or actually being built today; here we can only briefly outline one way to think about parallel computation effects and their impact, reserving more detailed discussion for section 2.4 of chapter 2.[18]

Importantly, it doesn't appear that parallel computers will affect our complexity results. NP-hard problems are still intractable on any physically realizable parallel computer. Problems harder than that are harder still. In brief, we can still use our complexity classification to probe grammatical theories.

This invariance stems from a fundamental equation linking serial (ordinary) computation time to the maximum possible speedup won via parallel computation. We envision a computer where many thousands of processors

[17] Of course, this does not rule out the possibility of a much more clever kind of preprocessing. It's just that no such examples have been forthcoming, and they all run the risk of destroying any close connection between the grammatical theory and language processing (if that kind of transparency is desirable).

[18] Chapter 2 briefly mentions the related topics of approximate solution algorithms but does not address yet another area of modern complexity—probabilistic algorithms—that might also shed light on grammatical formalisms. The end of chapter 2 also discusses the relevance of fixed-network "relaxation" neural models for solving hard problems, such as the neural model recently described by Hopfield and Tank (1986).

work together (synchronously) to solve a single problem.

$$\begin{array}{ccc} \text{Serial time} & & \text{Parallel time} \\ \text{to solve} & \leq & \times \\ \text{a problem} & & \#\text{ of parallel processors} \end{array}$$

This equation subsumes a wide range of examples. Suppose we have only a fixed number, k, of parallel processors. Our equation tells us that the best we could hope for would be a constant speedup. To do better than this requires a number of processors that varies with the input problem size.

Consider for example context-free language recognition; this takes time proportional to n^3, where n as usual is input sentence length. Suppose we had proportional to n^2 parallel processors; then our equation suggests that the maximum speedup would yield parallel processing time proportional to n. Kosaraju (1975) shows how this speedup can in fact be attained by simple *array automaton* parsers for context-free languages.

Using this equation, what would it take to solve an NP-hard problem in parallel polynomial time? It's easy to see that we would need more than a polynomial number of processors: because the left-hand side of the equation for serial time could be proportional to 2^n (recall that we assume that NP-hard problems cannot be solved in polynomial time and in fact all known solution algorithms take exponential time), and because the first factor on the right would be proportional to n^j (polynomial time), in order for the inequality to hold we could need an exponential number of parallel processors.

If we reconsider figure 1.1 in terms of processors instead of microseconds, we see that the required number of processors would quickly outstrip the number we can build, to say nothing of the difficulty of connecting them all together. Of course, we could build enough processors for small problems—but small problems are within the reach of serial machines as well. We conclude that if a grammatical problem is NP-hard or worse, parallel computation won't really rescue it.[19] We can rest secure that our complexity analyses stand—though we hope that the theory of parallel complexity can lead to even more fine-grained and illuminating results in the future.

[19]Section 2.4 of chapter 2 discusses certain problems that benefit from a superfast speedup using parallel processing; these include context-free language recognition, as mentioned (but probably *not* the corresponding universal context-free parsing problem); sorting; and the graph connectivity problem. This superfast parallel speedup may be closely related to the possibility of representing these problems as highly separable (modular) planar graphs.

1.5 An Outline of Things to Come

Having said something about what complexity analysis is and how it works, we conclude with a summary of what's to come: a more thorough look at complexity theory and its application to several grammatical systems.

Chapter 2: Before plunging into new and unfamiliar technical territory, any reader deserves an account of what to expect. Chapter 2 fills this need for readers unfamiliar with complexity theory. It sketches more formally the core concepts of complexity theory and the notation we'll use in the rest of the book. It also surveys the key problem transformations—like the one we used in our toy grammar example—that we'll use later on. Finally, it addresses in more depth the questions raised in chapter 1: whether our complexity theory idealizations are the right ones, including such topics as the effects of compilation and parallel computers.

Chapter 3: Agreement and lexical ambiguity are pervasive in natural languages: in English, subjects must agree with verbs in number and person, while many words like *kiss* can be either nouns or verbs. Chapter 3 defines a general class of agreement grammars (AGs) to formalize these notions, and shows that these two mechanisms alone suffice to make a grammatical formalism computationally intractable. Because of the simplicity of the result, *any* grammatical theory that incorporates agreement and lexical ambiguity—including most existing theories—will inherit the computational intractability of AGs. To resolve this dilemma, section 3.3 argues that this difficulty reflects a real possibility in natural language, and that performance theory truncations may be required to win efficient sentence processing.

Chapter 4: Lexical-functional grammar (LFG) has been proposed as a computationally more efficient grammatical formalism than transformational grammar (Kaplan and Bresnan 1982). Chapter 4 shows that LFGs contain enough agreement machinery and lexical ambiguity to inherit the intractability of AGs. Nothing in the LFG formalism, then, accounts for efficient human sentence processing; we need to supply an additional performance theory and/or new formal restrictions here. Chapter 4 proposes linguistic constraints—importing more $\overline{X}$ theory into the lexical-functional framework—as well as locality constraints to improve computational tractability.

Chapter 5: Most natural language processing systems have some way to decompose words into their parts so that they can be looked up in a dictionary and their constituent features returned; this precedes sentence analysis. Note that two factors are involved: dictionary lookup and spelling analysis are needed because a surface form like `tries` is retrieved from the dictionary as `try+s`. How hard is it to do this in general (and not just in English)? Chapter 5 probes this question by investigating one recently proposed model for dictionary analysis and retrieval—the *two-level* model of Koskenniemi (1983). Though the model is grounded on finite-state automata and gives an initial appearance of efficiency, two-level processing is in fact computationally intractable in the worst case; while some examples of spelling change–dictionary retrieval can be solved efficiently, not all of them can be. Chapter 5 shows why and underscores the point that simply relying on finite-state machinery need not save one from computational difficulties. It also shows that "compiling" a set of two-level automata into a single, larger one does not save us from computational trouble here.

Chapter 6: Chapter 5's analysis makes us suspect that there's something about the special properties of natural morphological systems that's not being exploited by the two-level model. Something less powerful than combinatorial search seems to be all that's required for analyzing words of English, Turkish, or other natural languages. To explore this possibility in a preliminary way, chapter 6 proposes to use constraint propagation for doing morphological analysis. Besides being demonstrably weaker than combinatorial search, constraint propagation seems to more accurately reflect the special features of local information flow and linear separability that seem characteristic of natural morphological analysis.

Chapter 7: Many modern linguistic theories cast surface sentence complexity as the result of several modular, interacting components. For example, the ID/LP formalism (Gazdar *et al.* 1985) separates out immediate dominance information (a sentence dominates an *NP* and a *VP*) from linear precedence (LP) information (a verb comes before an *NP* object). ID/LP systems have been proposed to partially describe so-called free-word-order languages, where noun phrases may appear in any order. Chapter 7 examines how hard it is to parse sentences generated by just the ID component of ID/LP grammars, dubbed *unordered context-free grammars* (UCFGs). While the literature suggests

that straightforward extensions of known parsing algorithms will work efficiently with these grammars, chapter 7 proves that this is not so: though writing down a free-word-order language in ID/lP form can often be beneficial, in the worst case, the sentences of an arbitrary ID system cannot be efficiently parsed. Here again the proof gives us some clues as to why natural free-word-order languages don't generally run into this difficulty, and suggests some natural constraints that might salvage computational tractability. Appendix A gives formal proofs for this chapter's claims.

Chapter 8: *Generalized phrase structure grammar* (GPSG), a recent linguistic theory, also seems to promise efficient parsing algorithms for its grammars, but this chapter shows that nothing in the formal framework of GPSG guarantees this. Modern GPSGs include a complex system of features and rules. While feature systems—simply saying that a noun phrase like *dogs* is singular and animate—may seem innocuous, much to our surprise they are not. It is an error to sweep features under the rug: the feature system of GPSG is very powerful, and this chapter shows that even determining what the possible feature-based syntactic categories of a GPSG are can be computationally difficult. Taken together, the components of GPSG are extraordinarily complex. The problem of parsing a sentence using an arbitrary GPSG is very hard indeed—harder than parsing sentences of arbitrary LFGs, harder than context-sensitive language recognition, and harder even than playing checkers on an $n \times n$ board. (See appendix B for some actual calculations of English GPSG grammar sizes.) The analysis pinpoints unnatural sources of complexity in the GPSG system, paving the way for the following chapter's linguistic and computational constraints.

Chapter 9: Drawing on the computational insights of chapter 8, this chapter proposes several restrictions that rid GPSGs of some computational difficulties. For example, we strictly enforce $\overline{X}$ theory, constrain the distribution of gaps, and limit immediate dominance rules to binary branching (reducing the system's unnatural ability to count categories). These restrictions do help. However, because revised GPSGs retain machinery for feature agreement and lexical ambiguity, revised GPSGs, like AGs, can be computationally intractable. Chapter 9 suggests this as a good place to import independently motivated performance constraints—substantive constraints on human sentence processing that aren't a part of the grammatical formalism.