A Mechanic's Guide to Grammar

Part I: Language in man and machine

By Brian Hayes

Good heavens! For more than 40 years I've been speaking prose and didn't even know it.
—M. Jourdain in Molière's Bourgeois Gentleman

We mock M. Jourdain for his belated discovery, but it seems to me his astonishment is entirely appropriate. Speaking prose is indeed an amazing feat. Like walking or dreaming or digesting, it falls into the peculiar category of things we do without knowing how we do them. A five-year-old can frame perfectly lucid English sentences, and yet no one has given a full and precise account of what constitutes a sentence. The algorithm for English is unknown, even to those who know English.

It is for just this reason—the lack of an algorithm—that we do not speak English to computers. The languages devised for communicating with machines are simpler and inevitably cruder; compared with English, a programming language such as Pascal has an extremely narrow expressive range. In one crucial respect, however, Pascal is on an even footing with English: both languages are infinite in scope. There is no limit to the number of English sentences, and so a reader or listener must always be prepared to cope with something new, with a combination of words no one has put together before. When you say, "Now I've heard everything," you are surely wrong. Likewise there are infinitely many Pascal programs, and the author of a compiler for Pascal cannot possibly anticipate all of them.

The key to taming these infinities is the idea of a grammar: a finite set of rules that describe an infinite set of sentences (or programs). Through grammar a language maintains an exquisite balance of novelty and regularity. Phrases never heard before can be instantly recognized as English; programs never written before can be compiled as efficiently as old standards.

In this series of articles I shall outline what grammar has to contribute to the understanding of language in man and machine. In Part I the emphasis is on properties common to all languages and all grammars. Next month I shall discuss a hierarchy of languages arranged according to the complexity of their grammars. Part III will explore the uses—and the limitations—of these ideas in compilers and other programs that deal with linguistic information.

Elements of language

The gulf between English and Pascal is so wide that it is fair to ask whether the term "language" can contain them both. A natural language has many voices and moods; it can be used to order breakfast, declare war, pray for peace, tell jokes, sell used cars, curse the umpire, grieve, rejoice, or describe the confused aspirations of a rich merchant in 17th-century France.

Programming languages are far less versatile. Many of them have only one mode, namely, the imperative "Do this, then this, then this." A program in such a language is a list of instructions, and the only form of natural language discourse it much resembles is a recipe. Indeed, the primary function of a programming language is not to communicate ideas (although it might often be used for that) but to elicit an action, to control a machine. English says, Pascal does. When the poet's work is done, we read the poem, but when the programmer's work is done, we do not read the source code; we execute it. These distinctions are of fundamental importance, but it does not follow from them that natural and formal languages have nothing whatever in common. "Language" can indeed be stretched around both English and Pascal although only by giving the word an extremely broad definition, one that encompasses much else besides—mathematical notation, numerals, bird songs, everything from the nucleotide triplets of DNA to Dante's terza rima.

The definition I have in mind is built from the bottom up. The first step is to specify an alphabet: a finite set of symbols. A sentence is defined as a string of symbols drawn from the alphabet. A grammar is a set of rules that determine whether or not any given string of symbols is an acceptable sentence. The language is the set of all sentences, that is, all strings of symbols accepted by the grammar.

It should be noted that nothing in this definition imputes any meaning to a sentence. The aim is to describe the syntax of the language, not its semantics. "Colorless green ideas sleep furiously" is non-

sense, but it is a sentence all the same. "Furiously sleep ideas green colorless" fails at a deeper, grammatical level.

For English the alphabet can be identified with the set of written characters or with the phonemes of spoken language. Alternatively, one can ignore this lowest level of structure and assume that the atomic units of language are words. Similarly, in a programming language the alphabet can be taken to consist of characters, or it can be the set of tokens, the higher-level constructs analogous to words, such as the keywords *if, then* and *while*.

What concept in a programming language corresponds to the English notion of a sentence? In Pascal one might be tempted to choose the individual statement, which seems to have roughly the same scale as a sentence, but I suspect the closest equivalent is a larger unit: the procedure or function definition. I shall not attempt to justify this choice, largely because it is of little consequence. In fact, no harm is done if an English sentence is made to correspond to an entire Pascal program. Ultimately, a sentence is whatever the grammar defines it to be.

And what is the nature of the grammar? Exploring that question is the purpose of this article, but before getting on with it one small warning must be posted. The grammars under discussion here have very little to do with the subject taught in grammar school. Lessons about dangling participles and parallel construction offer important guidance on the precise and graceful use of a language, but they cannot possibly capture its fundamental structure.

Generators and recognizers

The study of language has a long if somewhat sporadic history, going back to the first grammars written in Sanskrit before 300 B.C. It is an interesting history and is too little known, but I must pass over all of it, skipping directly to the 20th century, when the aims and methods of linguistic inquiry changed dramatically.

The change came about in part through the influence of work in other fields of inquiry: mathematics, logic, information theory and the study of automata. For example, in the 1930s and 1940s Alan M. Turing, Emil L. Post, Stephen C. Kleene and others investigated the properties of various abstract systems that qualify as languages under the definition given previously. Many of these loose threads were gathered up in the 1950s by Noam Chomsky of the Massachusetts Institute of Technology, who adapted the earlier work on formal systems to the study of natural language and added novel ideas of his own.

The goal of Chomsky's original program of research was a generative grammar. In its most abstract form the grammar is nothing more than a set of rules, but one can imagine it being embodied in a machine. The grammar machine for language L generates all the strings of symbols that are sentences of L and only those strings. The same set of rules can be incorporated into another machine, a recognizer rather than a generator. Given any string of symbols, the recognizer answers a yes-or-no question: Is the string a sentence of L?

For English any native speaker of the language can serve as a recognizer. To determine whether or not a string of English words is a sentence, merely present it to a native speaker of English and ask for an intuitive judgment. For a programming language a recognizer is also ready at hand: it is a component of any compiler or interpreter for the language. To find out whether a given string of Pascal tokens is a Pascal program, submit it to a Pascal compiler. If no error message is issued, the string passes the test.

Note that the compiler, in its role as recognizer, tests only the grammatical form of the program, not its semantic content. Nonsense programs, which give erroneous results or do nothing at all, are accepted without complaint as long as they have no syntactic errors.

For any finite language there is a trivial recognizer. It is a machine with a stored list of all the genuine sentences of the language. Whenever a new candidate string is proposed, the machine attempts to match it with each recorded sentence and reports its success or failure. This strategy cannot work for an infinite language. (In practice it is useless even for a language that is finite but large.)

The assertion that a language is infinite deserves careful scrutiny. If the alphabet of symbols is finite, the language can be infinite only if there is no limit on the length of a sentence. For English this is evidently the case. In the sentence "Neutrinos are very, very, very . . . very small" no limit can be put on the repetitions of "very." The sentence may grow absurdly cumbersome, so that you forget the beginning before you come to the end, but there is no definite boundary where the sequence of words ceases to be a sentence.

In the case of an infinite programming language other issues arise—or perhaps they are the same issues seen more clearly. Every real computer has only a finite amount of storage, and compilers too have limits on their capacity. These

Rudimentary grammar

1. S ::= NP VP
2. NP ::= Noun | Det NP | Adj NP | NP PP
3. VP ::= Verb | Verb NP | Verb PP | Verb Adv
4. PP ::= Prep NP

Rudimentary grammar could account for the structure of many simple English sentences (those with only a single clause). Each rule states that the symbol on the left of the ::= sign can be expanded into the sequence of symbols on the right. The | symbol, read "or," allows alternative productions to be included in a single rule.

Figure 1.

constraints are important, and they will be discussed in Part III of this series. For now it is enough to point out that they lie in the machinery of the recognizer, not in the definition of the language. There is nothing in the grammar of Pascal to forbid a program of, say, 10^{25} lines. All that is lacking is a floppy disk the size of the galaxy to hold the source code.

Of course no one has ever uttered an infinitely long sentence or written an infinitely long program. Moreover, if such a project should be undertaken, the recognizer could not render a verdict because it would never reach the end of the input. This might seem to undermine the notion of an infinite language, but it does not. There is no need to actually generate infinite sentences. It is enough to state that no matter how long a particular sentence is, we can always cite another sentence one word longer. The sentences of English are inexhaustible; all the fast-talking salesmen in the world cannot use them up.

Deep structure

A grammar for an infinite language requires rules that operate by a more ingenious mechanism than simple pattern-matching. The key, it turns out, lies in seeing a sentence as more than a string of words; it has, in Chomsky's terminology, a deep structure hidden beneath the one-dimensional surface structure.

The deep structure can be represented as a tree in which the leaf nodes are the words of the sentence and the organization of the interior nodes defines how the words are related to one another. The tree is generated or analyzed by repeated application of rules that have been variously named phrase-structure rules, rewrite rules and productions. The set of rules makes up the grammar.

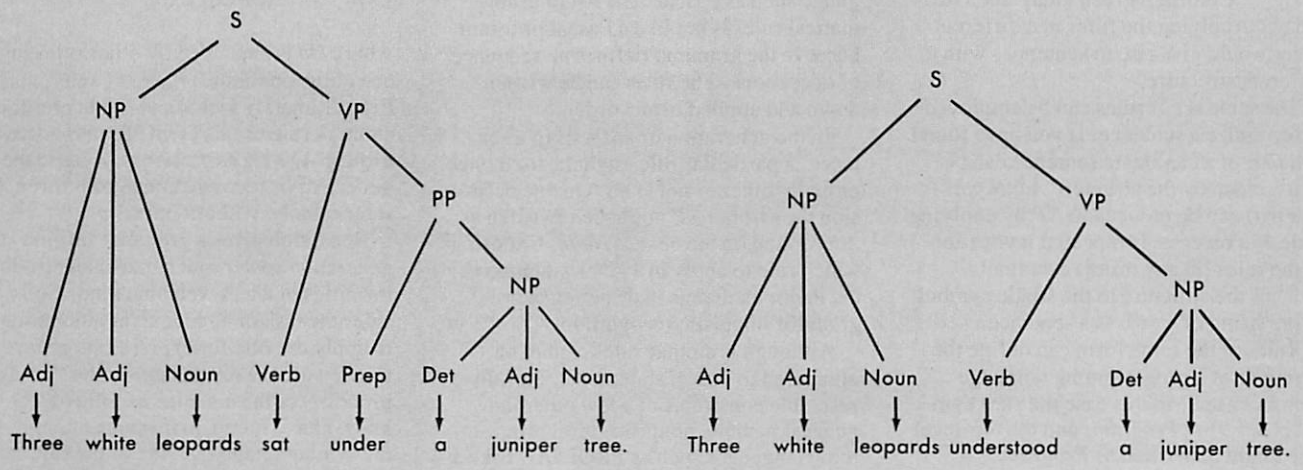
A bare-bones grammar for a subset of English can be given in four phrase-structure rules:

1. S ::= NP VP
2. NP ::= Noun
3. NP ::= Adj NP
4. VP ::= Verb Adv

Here the symbols *S* (for sentence), *NP* (noun phrase) and *VP* (verb phrase) are nonterminal symbols that correspond to the interior nodes of the tree. *S* marks the root node, although in complex sentences it can appear elsewhere as well. *Noun, Verb, Adj* and *Adv* stand for categories of words, which are the terminal symbols appended to the leaf nodes. Wherever the symbol *Adj* appears, an adjective is inserted, every instance of the symbol *Adv* is replaced by an adverb, and so on. The symbol *::=* has the meaning "can consist of" or "can be rewritten as." It should not be confused with the assignment operator *:=* in Pascal.

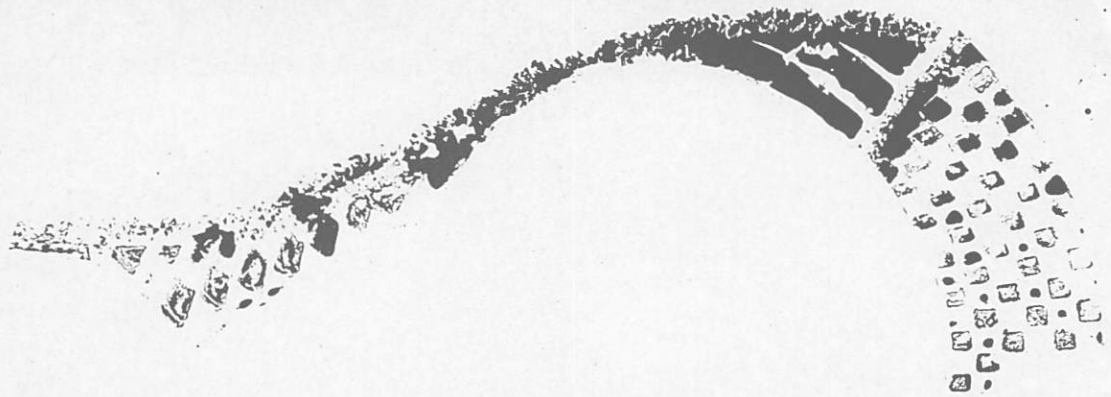
Rule 1 states that a sentence can consist of a noun phrase followed by a verb phrase. More formally, it states that whenever the symbol *S* is encountered in

Similar sentences with different deep structures



Tree diagrams for two English sentences reveal that they differ in deep structure despite similarities in surface structure. Nonterminal symbols of the grammar appear only on interior nodes of the tree, and terminal symbols only on the leaves.

Figure 2.



the structure of a sentence, it can be rewritten as *NP* followed by *VP*. Similarly, Rule 2 provides that a noun phrase is allowed to consist of a single noun. There is a second rule for the noun phrase, however; under Rule 3 *NP* might also be expanded to yield an adjective followed by a noun phrase. The verb phrase, according to Rule 4, produces a verb followed by an adverb.

The process of generating a sentence begins with the start symbol *S*, which appears only once in the grammar, on the left side of Rule 1. Applying this rule to *S* yields the structure *NP VP*. Expanding the noun phrase by applying Rule 3 yields *Adj NP VP*, and applying the same rule again leads to *Adj Adj NP VP*. Suppose the remaining *NP* symbol is then rewritten as *Noun* by Rule 2, and the verb phrase is converted into *Verb Adv* by Rule 4. The result is the structure *Adj Adj Noun Verb Adv*, which cannot be expanded further. None of the symbols appear on the left side of a rule.

Words of the appropriate categories can now be inserted in each position, giving rise to any one of a great many sentences: "Big, bad John wept bitterly." "Massless, chargeless neutrinos tunnel industriously." "Colorless green ideas sleep furiously." Applying the rules in a different order would give rise to sentences with a different structure.

The same set of rules can be employed to recognize a sentence. If you have found that part of a candidate sentence can be represented by the structure *Adj Noun*, that part can be reduced to *NP* by applying Rule 3 in reverse. If repeated invocation of the rules (in any order) eventually reduces the structure to the single symbol *S*, the string of words is a sentence.

Rules of the same form can define the grammar of a programming language such as Pascal. In this case the start symbol is not *S* but *Program*, and the top-level rule might be written as *Program ::= Header Block*. The latter symbol could be expanded by the rule *Block ::= DeclarationPart StatementPart*. Other productions would expand these symbols in turn and eventually would lead to the generation of terminal symbols for the tokens of the language.

I have expressed these rules in a variant of the notation called BNF, which originally meant Backus normal form. Here normal signifies that the rules have been written in a way that allows them to be handled easily by a particular kind of recognizer. John Backus of IBM, the leader of the group that created FORTRAN in the early 1950s, introduced the notation during the development of ALGOL-60. It is now generally called Backus-Naur form to recognize the contributions of Peter Naur of the University of Copenhagen, Denmark, editor of the ALGOL-60 report.

Extensions to BNF and certain shorthand conventions can make a grammar specification more concise. In particular, the symbol | , meaning "or," allows various alternative productions to be included within a single rule. Thus Rules 2 and 3 can be condensed into *NP ::= Noun | Adj NP*.

The rules of a grammar written in BNF look rather like the statements of a programming language. The resemblance is misleading, at least if one has in mind an ordinary, imperative programming language such as Pascal. The list of grammatical rules is not like a Pascal program because the grammar defines no sequence of operations. The rules can be written down and applied in any order.

In the generation or analysis of a sentence, a particular rule might be used once or many times or not at all. On one occasion the symbol *NP* might be rewritten as *Adj NP* and on the next as *Noun*. Choosing which rule to apply in a given situation is the major challenge in designing a program for linguistic recognition.

Although grammar rules cannot be compared to Pascal statements, they do resemble constructs of a few other languages, namely, nonprocedural programming languages such as PROLOG. Like a grammar, a PROLOG program consists of statements that can be invoked in any order. The statements are not instructions to be executed but declarations of facts or relations. Because of this correspondence it is particularly easy to write a program for a recognizer in PROLOG; the program is little more than a specification of the grammar. SNOBOL provides similar facilities.

Recursion

How can a grammar made up of a finite number of production rules generate an infinite language? The source of this power is recursion. The potentially endless series of adjectives generated by Rule 3 represents one form of recursion. A more elaborate recursive scheme is at work in the pair of rules:

```
NP ::= Noun | Noun PP
PP ::= Prep NP
```

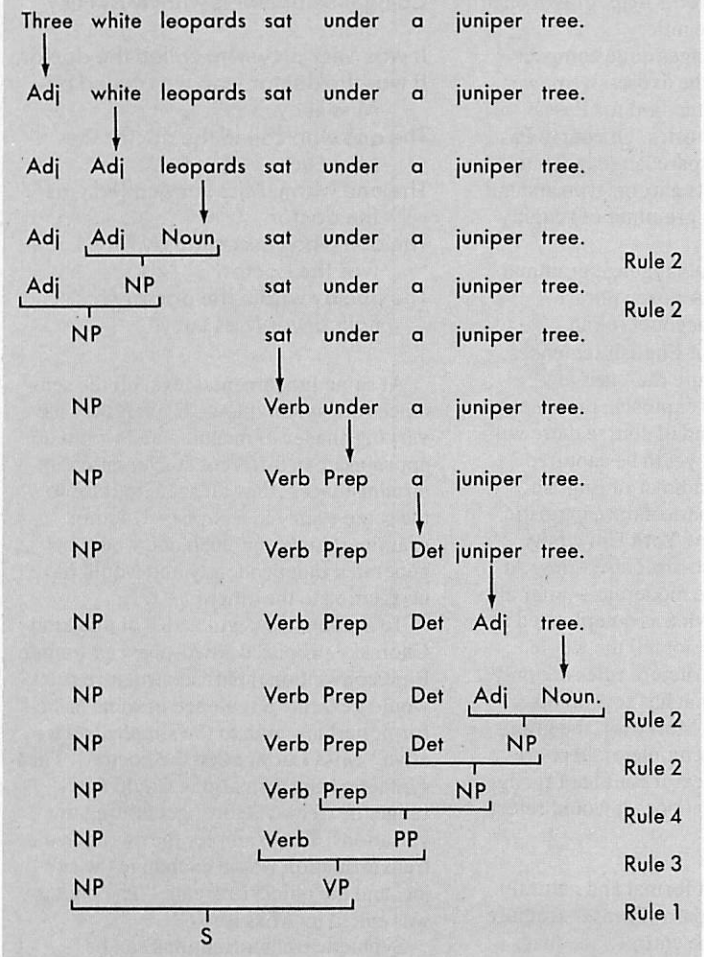
Here a noun phrase is defined so that it can include an optional prepositional phrase (*PP*), and the prepositional phrase in turn includes a noun phrase. Each time *NP* is expanded into *Noun PP*, the subsequent expansion of *PP* gives rise to another instance of *NP*. The interaction of the two rules can create an infinite cascade of prepositional phrases, as in "the hair of the dog in the lap of the man under the table in the house on the cliff by the sea . . . " Recursive rules also appear in the grammar of any nontrivial programming language. One entry in a set of rules defining arithmetic expressions might read:

```
Expr ::= Expr Op Expr
```

where *Op* is expanded by other rules into one of the operators +, −, ∗, and /, and *Expr* ultimately yields a variable or a constant. A rule of this form accommodates arbitrarily long expressions because the generator or recognizer can pass through it repeatedly, without limit.

Recursion gives a grammar infinite generative power, but it can also introduce infinities of a less welcome kind. Consider the task of a recognizer attempting to apply the rule for expressions given previously. We can personify the "thought process" of the machine as follows: "I know I have found an instance of *Expr*, the symbol on the left side of the rule, if I can find in the input string a sequence of symbols that matches what is on the right side of the rule. The right side begins with *Expr*, and so the first step is to find a symbol that could be an expansion of *Expr*. To

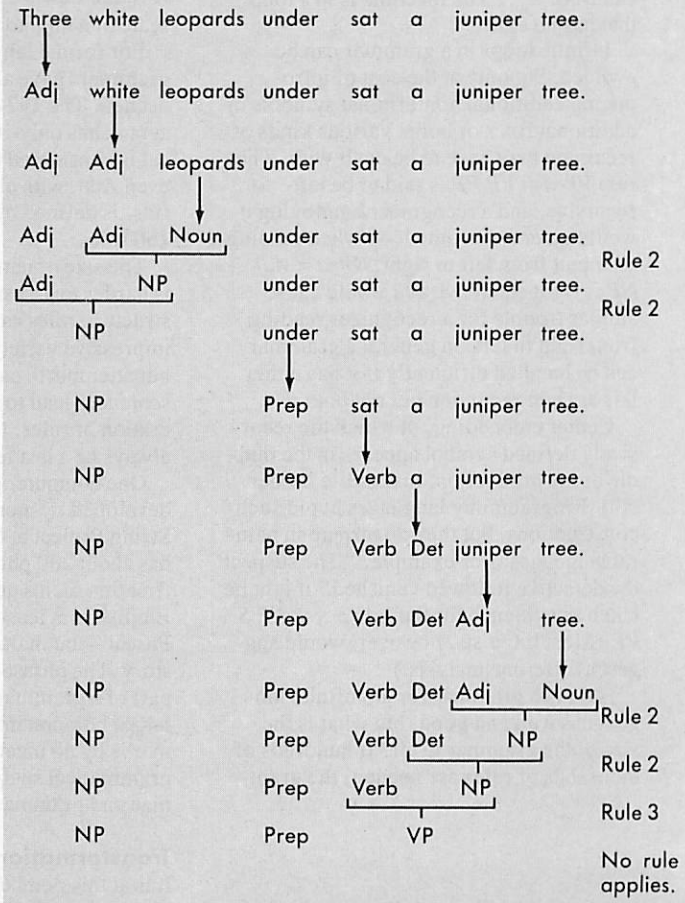
Successful recognition of a sentence



Recognition procedure employs the grammar given in Figure 1 to confirm that a string of words is indeed a sentence. First each word, reading from left to right, is replaced by its syntactic category, such as adjective or noun. Whenever a sequence of symbols matches a pattern on the right side of a rule, the sequence is replaced by the symbol from the left side. Ultimately the entire structure is reduced to the symbol *S*. The order in which the rules have been applied here is not the only one that leads to successful recognition.

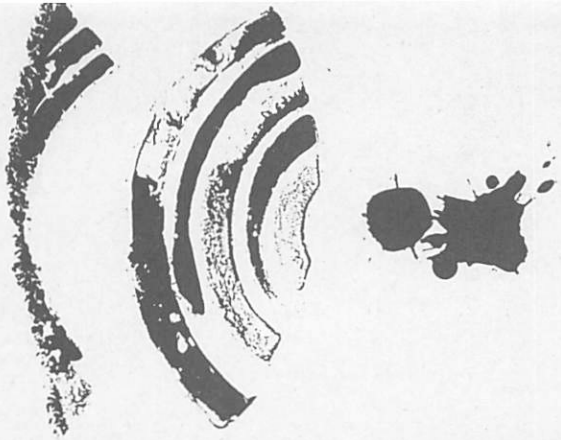
Figure 3.

An attempt to recognize a nonsentence fails



Nonsentence is rejected as ungrammatical by the recognition procedure. The analysis begins as in Figure 3 but gives rise to the structure *NP Prep VP*; no part of this pattern matches the right side of a rule, and so the attempt at recognition ends in failure.

Figure 4.



do that, I can apply the same rule again. I know I have found an instance of *Expr* if I can find. . . ." The machine is in a loop that has no exit.

Infinite loops in a grammar can be avoided, but only at the cost of introducing additional nonterminal symbols or additional rules or both. Various kinds of recursion may have to be dealt with. The rule *VP ::= VP PP* is said to be left-recursive, and a recognizer employing it would enter an infinite loop when reading the input from left to right. *NP ::= Adj NP* is right-recursive and would cause similar trouble for a recognizer reading from right to left. In general a grammar can be handled efficiently if it has either left or right recursion but not both.

Center embedding, in which the recursively defined symbol appears in the middle of the production, makes life harder still. Programming languages avoid such constructions, but they do turn up in natural languages. For example, "The suspect the detective followed vanished" might be taken as evidence for the rule *S ::= NP S VP.* (Most linguists, however, would suggest a different analysis.)

Having a grammar for an infinite language is well and good, but what is the size of the grammar itself? If hundreds of thousands of rules are needed, the grammar is no more accessible to human comprehension than the language it describes. It is like Lewis Carroll's map, drawn on a scale of a mile to the mile.

For formal languages quite compact grammars have all the expressive power needed. The 1974 standard for Pascal syntax has only 107 rules. Of course Pascal is considered a spare language, but even Ada, with all its chrome trim and tail fins, is defined by a grammar of roughly 150 rules.

The size of natural language grammars is harder to assess. A dozen phrase-structure rules can account for an impressive variety of English sentences, but attempts to capture the "last few" sentences lead to an explosive proliferation of rules. (And of course there will always be a last few yet to be captured.)

One computer grammar of English, developed by members of the Linguistic String Project at New York University, has about 230 phrase-structure rules. At first this seems quite moderate— after all, English is at least twice as complicated as Pascal— but it does not tell the whole story. The phrase-structure rules are only part of a grammar that has several other, larger components. Moreover, the grammar is by no means complete; there are grammatical sentences it could not recognize and grammatical ones it would reject.

Transformations

It is at this point that formal and natural languages part company. Phrase-structure grammars seem to be entirely adequate for programming languages and other invented notations, but something more is needed to describe the sentences people speak. Conceivably a comprehensive list of production rules could be compiled for English—as far as I know, there is no proof one way or the other—but at best it would be a long, clumsy, and unilluminating document.

One notable deficiency of a phrase-structure grammar is that it offers no natural explanation of sentences that appear to be related even though they differ in both deep and surface structure. Consider the following series of sentences:

Miss Lucy called the doctor.
The doctor was called by Miss Lucy.
What Miss Lucy did was call the doctor.
Calling the doctor is what Miss Lucy did.
It was Miss Lucy who called the doctor.
It was the doctor who was called by Miss Lucy.
The one who called the doctor was Miss Lucy.
The one whom Miss Lucy called was the doctor.
The one who was called by Miss Lucy was the doctor.
The one by whom the doctor was called was Miss Lucy.

At some fundamental level all the sentences are synonymous. Even if they have varying shades of meaning and would be appropriate in different conversational circumstances, they all seem to refer to the same event. In a phrase-structure grammar, however, each one would be generated independently and would have no relation to the others.

To account for regularities of this kind Chomsky proposed a two-phase grammar. First conventional phrase-structure rules would generate a sentence in some basic form, perhaps akin to the simple declarative "Miss Lucy called the doctor." Then syntactic transformations would rearrange the tree structure, generating the variations. For example, the passive voice transformation would exchange the subject and the object to create "The doctor was called by Miss Lucy."

Syntactic transformations can be defined in terms of the tree structure, without reference to the meaning of words. In this case the subject can be identified unambiguously as the noun phrase immediately dominated by the root node *S*, and the object is the noun phrase immediately dominated by a verb phrase that in turn is immediately dominated by *S*.

Transformational grammar has not turned out to be the ultimate and eternal theory of language, but that has surprised no one. Newtonian mechanics is not the ultimate theory of physics, and yet it is not disparaged on that account. In both cases,

Production rules

1. Stmt ::= Variable := Expr
2. Expr ::= Term | Term AddOp Term
3. Term ::= Factor | Factor MultOp Term
4. Factor ::= Variable | Constant | (Expr)
5. AddOp ::= + | −
6. MultOp ::= * | /

Programming language can also be defined by a grammar made up of production rules. The fragment shown here describes assignment statements and arithmetic expressions in a syntax like that of Pascal. The rules could be expressed more concisely and with fewer nonterminal symbols if it were not for the need to avoid infinite loops in the recognizer. Additional rules are needed to expand *Variable* and *Constant* into terminal symbols.

Figure 5.

where the theory works, it works well.

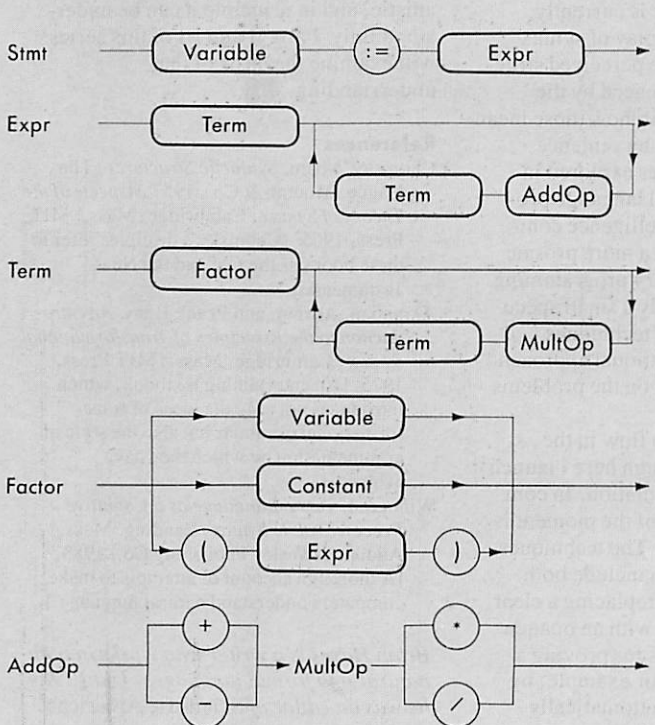
Within a limited domain transformational grammars offer an appealing parsimony: a vast corpus of sentences can be generated by a few dozen phrase-structure rules and an equal number of transformations. The question is whether the theory can be extended to take in those last few sentences without losing its elegance. Chomsky himself and many of his colleagues have lately advocated drastic revisions, putting less emphasis on transformations and more on phrase structure. There is certainly no shortage of problem sentences that do not fit into the scheme.

One problem common to many grammars, transformational and otherwise, has to do with lexical insertion. I mentioned previously that when a tree has been generated, words from the appropriate lexical categories are hung on the leaf nodes. This description of the process is far too vague to serve as the basis of a mechanistic theory. Exactly how are the words to be selected?

Trying to create an algorithm for lexical insertion quickly reveals that the standard categories of noun, verb, adjective, and so on are not nearly precise enough. Cookie and bread are both classified as nouns, but whereas "I want a cookie" is perfectly good English, there is something the matter with "I want a bread."

To keep track of such distinctions linguists posit a mental lexicon, where each word is entered along with a list of attributes that determine where it can appear in a sentence structure. (The distinction operating here is that cookie is a countable noun and bread an uncountable one.) As the categories of words proliferate, how-

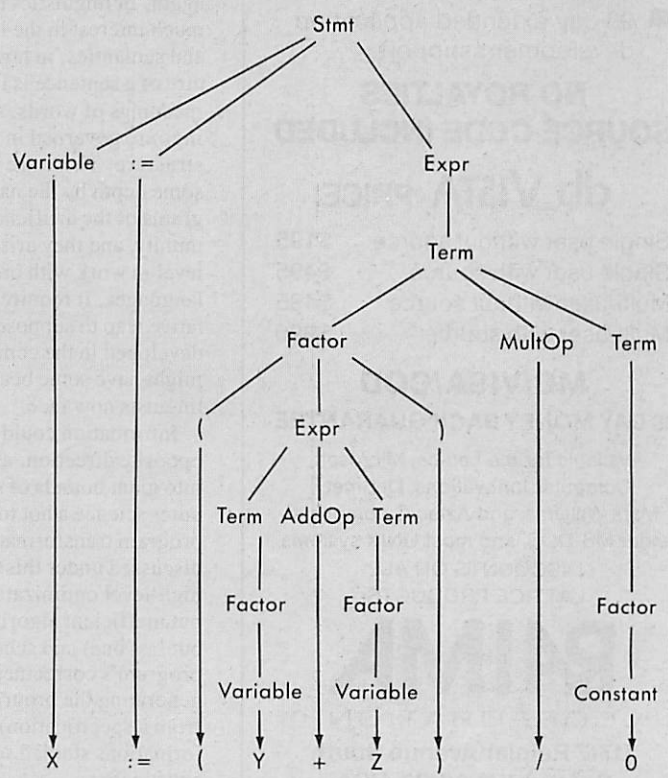
A syntax diagram



Syntax diagram, or recursive transition network, expresses the same information as the set of rules in Figure 5. Each nonterminal symbol can be expanded by following a path through the diagram. Branches allow for alternative productions, and loops provide for repetitive constructs.

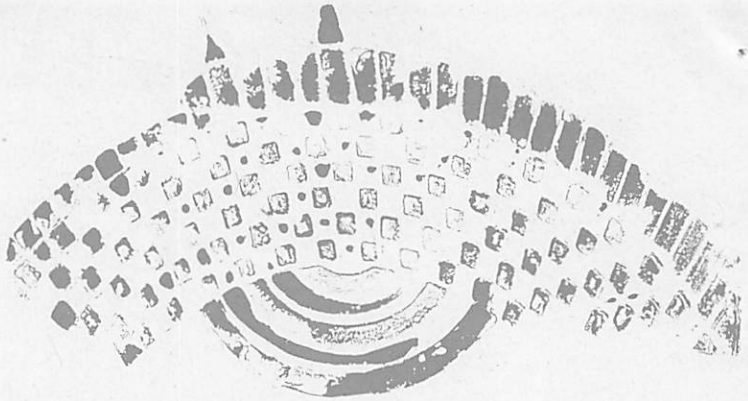
Figure 6.

A grammatical but meaningless statement



Simple assignment statement has a complex syntactic structure. Note that the statement is grammatical even though it is meaningless. (The expression calls for division by zero and hence is undefined.)

Figure 7.



ever, one begins to wonder if the elegance of the basic grammar is not being maintained simply by pushing all the complexity into the lexicon.

Cross currents

Thirty years ago Chomsky brought together ideas from the study of formal and natural languages, but the two lines of work diverged again almost immediately. Linguists dismissed the formal grammars as inadequate to account for the structure of natural language. By the same token, computer scientists designing programming languages or writing compilers have found little use for transformational grammars. Even work on computer interpretation of natural language has rarely drawn on the transformational principles introduced by linguists.

The two streams may yet converge again. In linguistics there is currently much interest in the interplay of syntax and semantics, in how the perceived structure of a sentence is influenced by the meanings of words, and in how those meanings are governed in turn by sentence structure. These are issues explored in some depth by the natural language programs of the artificial intelligence community, and they arise on a more prosaic level in work with ordinary programming languages. It requires only a small speculative leap to suppose the techniques developed in the computational approach might have some bearing on the problems linguists now face.

Information could also flow in the opposite direction, although here I launch into giant bounds of speculation. In computer science a hot topic of the moment is program transformation. The techniques discussed under this term include both high-level optimization (replacing a clear but inefficient algorithm with an opaque but fast one) and schemes for proving a program's correctness (for example, by generating the program automatically from a specification). The transformations studied so far come from logic and algebra.

Linguists have invested much effort in building a complex web of interdependent transformations and deducing the fixed order in which they must be applied. An interesting subset of those transformations are guaranteed to preserve the meaning of the underlying sentence, which is just the property demanded of program transformations. We are a long way from having a programming language defined by a transformational grammar, but the idea seems worth exploring.

Whether or not linguists and computer scientists can be of any use to each other, natural and formal languages remain coupled like shadow and light. The quandary in trying to understand how the mind recognizes natural language is that the recognizer itself is a machine of daunting complexity. If a feature of language seems mysterious, one can always appeal to some unknown process in the dark corners of the mind. With a language recognized by a computer there can be no recourse to mental hocus-pocus. What happens in the CPU is certain to be mechanistic, and in principle it can be understood fully. Parts II and III of this series will examine the basis of that understanding.

References

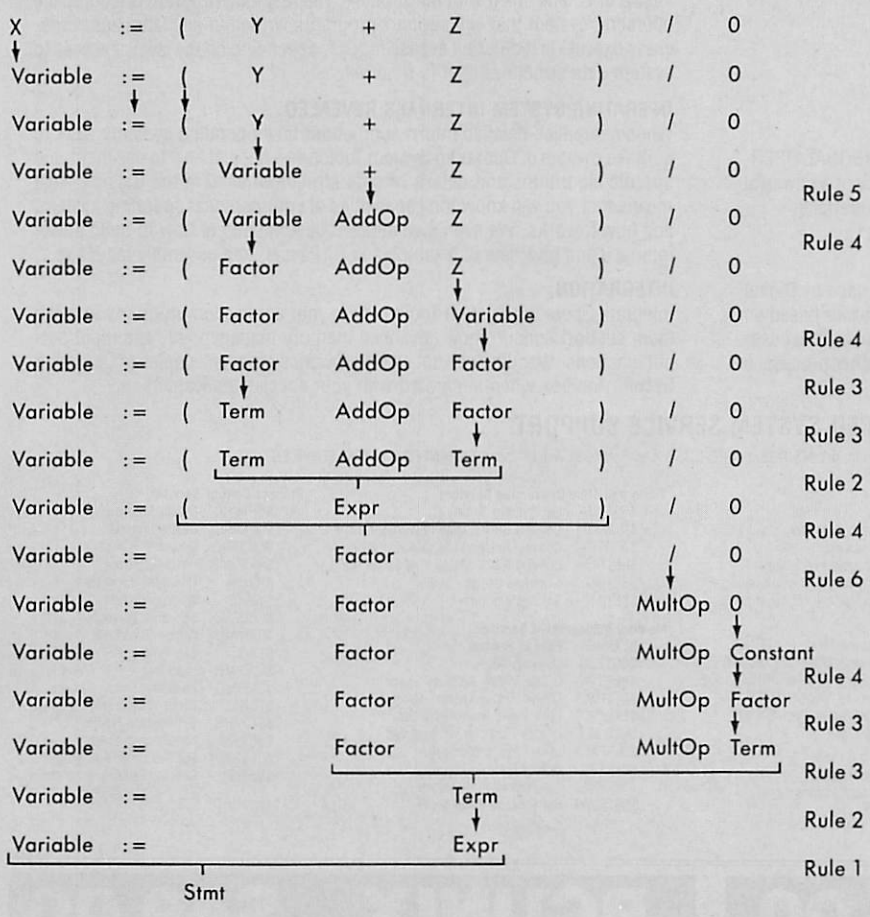
Chomsky, Noam. *Syntactic Structures*. The Hague: Mouton & Co., 1957. *Aspects of the Theory of Syntax*. Cambridge, Mass.: MIT Press, 1965. [Chomsky's disciples refer to these books as the Old and the New Testaments.]

Akmajian, Adrian, and Frank Heny. *An Introduction to the Principles of Transformational Syntax*. Cambridge, Mass.: MIT Press, 1975. [An entertaining textbook, which introduces not only the ideas of transformational grammar but also the style of argumentation by which they have evolved.]

Winograd, Terry. *Language as a Cognitive Process. Vol. 1: Syntax*. Reading, Mass.: Addision-Wesley Publishing Co., 1983. [A thorough account of attempts to make computers understand natural language.]

Brian Hayes is a writer who works in both natural and formal languages. Until 1984 he was an editor of Scientific American, *where he launched the Computer Recreations department.*

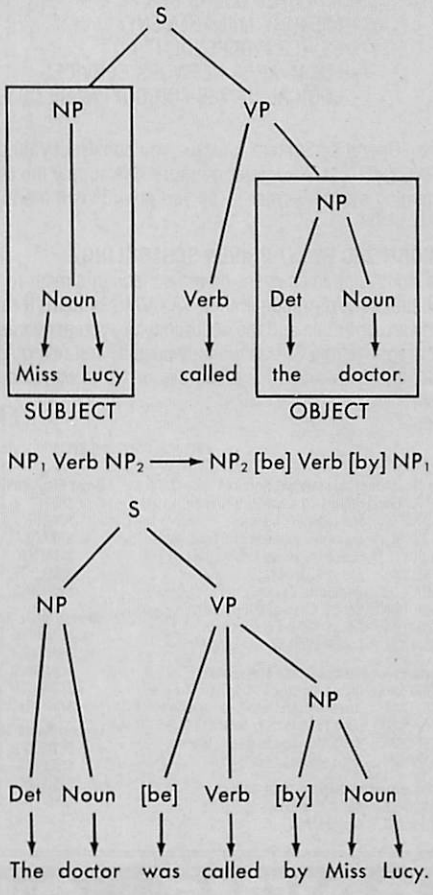
Steps in the recognition of a statement



Analysis of a program statement succeeds by reducing it to the single symbol *Stmt*. The input is read from left to right and a production rule is applied as soon as any pattern matching its right-hand side is encountered.

Figure 8.

A transformation



Syntactic transformation converts a sentence from the active to the passive voice while preserving its meaning. The transformation can be applied only to a sentence that matches the pattern NP_1 *Verb* NP_2 where the two noun phrases can be identified with the traditional syntactic categories of subject and object. The transformation exchanges the noun phrases and inserts a form of the verb be and the word by.

Figure 9.