A Mechanic's Guide to Grammar

Part II: Climbing the Tower of Babel

By Brian Hayes

After the fiasco at Babel, according to tradition, the primordial speech of Adam was shattered into 72 fragments. By the time anyone got around to counting, the situation had become a good deal worse: there were thousands of languages. The philologists of the 19th century set about cataloguing and classifying them and reconstructed the genealogy of several hundred. For example, they showed that English and Sanskrit are distant leaves of the same tree, whereas Finnish and Hungarian are members of a much smaller family, and Basque stands quite alone.

In recent years the situation has become still more complicated. Languages are no longer merely discovered; they are being invented at a frantic pace. Moreover, the notion of what constitutes a language has been expanded to take in notational systems (including programming languages) that 19th-century linguists would not even have recognized as proper to their field of study.

The greater scope given to the concept of language has brought with it new principles of analysis and classification, which owe more to mathematics than to traditional linguistics. The invented, formal languages are classified not according to their parentage but according to their structure or complexity. Etymology is irrelevant; what matters is the format of grammatical rules and the nature of the machine one would need to recognize the sentences of a language. From this formal analysis has come a new Tower of Babel: a hierarchy of language classes. It is called the Chomsky hierarchy, after Noam Chomsky of the Massachusetts Institute of Technology.

In some respects the Chomsky hierarchy conforms to common intuitive rankings. As one might expect, a language higher in the scale has a more complex grammar and requires a more elaborate recognizing machine. But a paradox lurks here. In saying one language is more powerful than another, we generally have in mind that it can be used for saying more, that it has a broader expressive range. Some such judgment surely lies behind the universal acknowledgement that English is more powerful and versatile than Pascal. In the Chomsky hierarchy, however, the languages at the top of the scale are not those that include the most. On the contrary, they are those from which the most is excluded.

Minimal languages

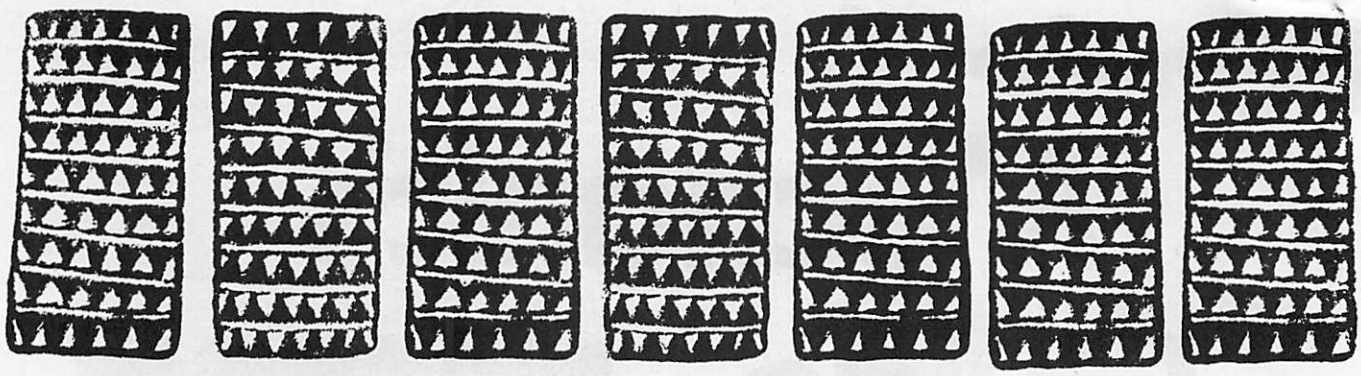
Last month, in Part I of this series, I discussed some of the relations between a language, its grammar, and the machines that can embody the grammar. This month I shall focus on what distinguishes one class of languages from another and on how the properties of a grammar determine the capabilities of the corresponding language.

Formal linguistics defines a language as a set, in the mathematical sense: a collection of unduplicated elements that satisfy some stated rules of membership. The elements of the set are strings of symbols, where the symbols in turn are drawn from a finite alphabet (another set). The selection rules are the grammar of the language. They determine whether or not any given string of symbols is a member of the set.

This definition clearly embraces all the natural languages of the world and all programming languages, but it includes a great deal else besides. Indeed, virtually anything that can be viewed as a sequence of symbols qualifies as a language. It need not have any evident meaning or use in communication. The numbers on a license plate form a sentence in some language; so do the sequence of stitches in a knitted sweater or the sequence of moves in a game of chess.

In exploring the realm of abstract languages the place to begin is at the bottom of the ladder. What is the simplest possible language? Since a language is defined as a set, there is a straightforward answer: it is the empty set, represented by the symbol $\emptyset$. And the next-simplest language consists of exactly one string, which happens to be made up of zero symbols. This empty string is denoted ϵ (the Greek letter epsilon). Note that $\emptyset$ and ϵ are distinct; one is a set without members, whereas the other is a set whose one member is empty.

After these two variations on languages that contain nothing, the next language is one that contains everything. It is a language with a trivial grammar, namely, a set of rules that admit any possible string made up of symbols from the defined alphabet.



Suppose the alphabet consists of the single symbol *a*. Then the all-encompassing language is the set of all strings composed of any number of *a*'s, including zero *a*'s. The strings are ϵ, *a*, *aa*, *aaa*, and so on. For the alphabet made up of *a* and *b* the corresponding language includes ϵ, *a*, *b*, *aa*, *ab*, *ba*, *bb*, *aaa*, etc. If the alphabet is the ASCII character set, the language includes essentially all of English and most programming languages. The trouble is, it includes vast tracts of gibberish as well. It is the language of Sir Arthur Eddington's army of monkeys drumming on the keyboard.

Regular sets and expressions

All of these languages are members of the class called regular languages or regular sets. The work that led to their definition was begun in the 1940s by Warren S. McCulloch and Walter Pitts, who were studying simple models of nerve cells. The connections with set theory and formal linguistics were established a decade later by Stephen C. Kleene of the University of Wisconsin at Madison.

Which languages are regular? Kleene defined the class by means of a notational system called regular expressions. Any language that can be described by a regular expression is a regular language.

In building a regular expression, three operations are allowed: concatenation, alternation, and repetition. Two symbols are concatenated by writing them one after the other: the concatenation of *a* and *b* is *ab*. Moreover, if *X* and *Y* represent strings of symbols, the concatenation *XY* denotes the set formed by writing any substring from *X* followed by any substring from *Y*. The symbol " | " signifies alternation. The expression $a \mid b$ allows a choice of either *a* or *b*, and $X \mid Y$ represents either a substring from *X* or one from *Y*. The repetition operator (known technically as the Kleene closure) is the asterisk. The expression a^* describes all strings made up of any number of *a*'s, including zero. X^* is the set formed by concatenating any number of substrings chosen from *X*.

Given these three operations, a regular expression can be defined recursively. First, $\emptyset$ and ϵ are admitted as regular expressions, and so is any individual symbol from a specified alphabet. The recursive part of the definition states that if *X* and *Y* are any regular expressions, then *XY*, $X \mid Y$, and X^* are also regular expressions. Nothing else is a regular expression.

A few examples serve to illustrate how a regular expression can describe a language. The set of all strings of zero or more *a*'s is represented by the regular expression a^*, and all strings of *a*'s and *b*'s are given by $(a \mid b)^*$. A different set is obtained from a^*b^*: it is the subset of $(a \mid b)^*$ in which any number of *a*'s can be followed by any number of *b*'s, but once the first *b* has appeared there can be no more *a*'s. Thus ϵ, *a*, *b*, *aab*, and *abbbbb* are all sentences of the language described by a^*b^*; but *aba* is not.

Can languages constructed on this model be of any practical use? Suppose *D* is the alphabet of digits from 0 through 9; then D^* is the language whose strings can be interpreted as integers. Actually, D^* includes the empty string ϵ, which is not a well-formed integer, and so it is better to specify the language as DD^*, ensuring that every number has at least one digit. The more elaborate expression $(+ \mid - \mid \epsilon)DD^*$ allows an integer to be preceded by an arithmetic sign, and $(+ \mid - \mid \epsilon)DD^*.D^*$ makes provision for floating point numbers. Further extensions for exponential notation, hexadecimal numbers, and so on are straightforward.

The identifiers that serve in many programming languages as names of variables, procedures, and functions derive their form from regular expressions. In ALGOL an identifier is a letter followed by any sequence of letters and digits. If *L* is the alphabet of letters, an ALGOL identifier is a member of the set specified by $L(L \mid D)^*$. Editors and other text-processing systems also make extensive use of regular expressions. The UNIX utility grep (an abbreviation of "globally look for regular expressions and print") searches a file for strings that match a description given as a regular expression.

The "wildcard" notation for file names in several operating systems has a similar format. The meaning of an asterisk is slightly different—it signifies closure over the entire alphabet rather than repetition of the preceding symbol—but the principle is the same. It is because of the properties of regular expressions that A^* finds all file names beginning with *A*, but $^*Z$ does not select names ending with *Z*.

Grammars and machines

The regular languages are at the bottom of the Chomsky hierarchy, but that does not mean they are the least important class. On the contrary, they are the foundation on which all else is built. The languages higher in the ranking are progressively more refined subsets of the regular languages.

Chomsky described the hierarchy in a series of papers written in the late 1950s. He identified four levels of complexity. Moreover, he showed that each class of languages is associated with a class of grammars and a class of machines. The regular languages are labeled type 3. Going up the ladder, through types 2, 1, and 0, the grammars become more complex and the machines more powerful while the languages are bound by ever tighter constraints.

The grammar of a formal language is a series of production rules, which state how one string of symbols can be rewritten to generate another string. The rule $XY ::= abc$ says that the string *XY* can be replaced wherever it appears by the string *abc*. A string may have more than one possible expansion; the alternatives are indicated by separating them with the symbol " | ". I shall adopt the convention that capital letters are nonterminal symbols, which never appear in the sentences of the language, and terminal symbols are lowercase. There is one special symbol, *S*, which starts the derivation of every sentence.

If a language is regular, its grammar can be expressed entirely in rules that satisfy stringent formal specifications. The left side of each rule must be a single sym-

bol, which is necessarily a nonterminal. The right side can consist either of a single terminal or of a terminal followed by a single nonterminal.

It can be proved that regular expressions and regular grammars describe the same class of languages, but here I shall merely suggest the nature of the correspondence by giving examples. The language specified by the expression a^* is generated by the grammar $S ::= \epsilon \mid aS$. Each time the second production is chosen, an a is added to the string; since the rule can be invoked repeatedly, any number of a's can be generated. The first alternative allows for the empty string.

The grammar can be made to generate all strings of a's and b's, like the regular expression $(a \mid b)^*$, by adding one more production: $S ::= \epsilon \mid aS \mid bS$. The grammar for a^*b^* calls for at least two rules:

S ::= ϵ | aS | bX
X ::= ϵ | bX

Once the production $S ::= bX$ is chosen, the grammar is committed to generating only b's.

The correspondence between languages and machines is perhaps the most intriguing outcome of Chomsky's inquiry into formal linguistics. It links the abstract realm of symbol systems to the concrete world of mechanisms. We can "see" how a language works by tracing the operation of the machine that recognizes its sentences.

For a regular language the recognizing machine is a finite-state automaton, or FSA. As the name suggests, the machine has a finite number of states, which must be discrete, or discontinuous. In each state the machine responds to some set of

Classes of grammars and languages

Type	Description	Form of rules	Recognizing machine	Example grammar	Example language	Comments
0	Unrestricted grammars; recursively enumerable languages	Left- and right-hand sides may consist of any strings of symbols	Turing machine	S : : = UXaV Xa : : = aaX XV : : = YV I Z aY : : = Ya UY : : = UX aZ : : = Za UZ : : = ϵ	a^{2^n} aa, aaaa, aaaaaaaa, aaaaaaaaaaaa aaa . . .	Can calculate any computable function
1	Context-sensitive grammars and languages	Left-hand side may include multiple symbols, but no more than one symbol is expanded on right-hand side	Linear-bounded automaton	S : : = aXY I aSXY YX : : = XY aX : : = ab bX : : = bb bY : : = bc cY : : = cc	$a^nb^nc^n$ abc, aabbcc, aaabbbccc, aaaabbbbcccc . . .	Can count three or more things but cannot calculate
2	Context-free grammars and languages	Left-hand side consists of a single symbol	Pushdown automaton	S : : = ab I aSb	a^nb^n ab, aabb, aaabbb, aaaabbbb aaaaabbbbb . . .	Can count two things but not three
3	Regular grammars and languages	Right-hand side has either one terminal symbol or a terminal followed by one nonterminal	Finite-state automaton	S : : = ϵ I aS I bX X : : = ϵ I bX	a*b* ϵ, a, b, aa, ab, bb, aaa, aab, abb, bbb, aaaa, aaab, aabb . . .	Cannot count

Chomsky hierarchy of languages consists of four classes. As one proceeds from the bottom of the scale to the top, the grammars are subject to fewer constraints and the machines needed to recognize a language become more powerful. The languages themselves, on the other hand, are defined by ever tighter strictures on the allowable form of a sentence.

Figure 1.

inputs, such as symbols from a finite alphabet. In many cases an input causes a transition to another state. There may also be outputs associated with some states. In the machines described here an output is issued only when the machine enters a final, or accepting, state, having recognized a sequence of inputs as a sentence in a language.

A simple FSA is shown in Figure 2. It has two states, labeled Even and Odd, and accepts as inputs the symbols 0 and 1. The machine begins operation in the Even state. On a 0 input it remains in the same state, but on a 1 it makes a transition to the Odd state. In this new condition a 0 again has no effect, but a 1 causes a transition back to the Even state. The machine detects the parity of a stream of binary digits; if Odd is designated an accepting state, the automaton will recognize all strings of 1's and 0's that have an odd number of 1's.

A crucial property of an FSA is that it has no auxiliary memory. It can keep track of the history of its inputs only by moving from one state to another. The machine's action at any moment is determined entirely by the present state and the present input. Earlier inputs have an influence only by helping to determine the current state.

The finite-state model can be applied to a great many systems, including some that appear to be remote from language theory. Electronic-logic networks made up of gates and flip-flops act as FSAs, and the equivalence is sometimes exploited in designing the networks. The language of nucleotide bases in RNA is recognized by an FSA called the ribosome, whose outputs are the amino acids of a protein. Other examples include light switches, vending machines, combination locks, and the quantum-mechanical model of the atom. The push-button mechanism of a ballpoint pen is an FSA that records the parity of its inputs. Figure 3 shows an FSA that recognizes the language of telephone numbers.

The connection between regular languages and FSAs implies that any finite language must be regular. After all, if the language has only a finite number of sentences, one could build an FSA with a state for each sentence. As it happens, many regular languages of practical importance are indeed finite. Identifiers and file names almost always have some limit on their length, whether or not it is stated explicitly. Nevertheless, the usual practice is to treat such languages as if they were infinite. (MS-DOS allows more than 10^{18} possible file names, and so the strategy of providing a state for each name is not attractive.)

What is more important, an FSA can recognize a language that is genuinely infinite. Even though the machine has only a finite number of parts, it can distinguish between grammatical and ungrammatical strings of unlimited length. The parity-testing machine is an apt example. It "knows" whether the number of 1's is even or odd no matter how long the input string becomes.

The key to this infinite capacity is that an FSA can have a loop in its structure, so that the machine passes through some group of states repeatedly. This observation is the basis of a fundamental result in language theory called the pumping lemma. It states that any potentially infinite string in a regular language must have a substring that can be "pumped," or repeated indefinitely. (The repeatable substring is just the sequence of symbols that causes the machine to pass through the loop.) A corollary is that a regular expression describing an infinite language must include at least one asterisk, and a grammar for the language must have at least one recursive rule, in which a nonterminal from the left side of a production also appears on the right side.

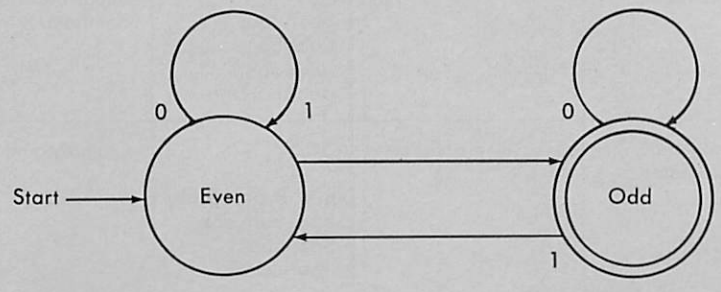
Context-free languages

If a regular language can encompass an infinity of sentences, why is there any need for more elaborate languages? The answer, in its simplest form, is that an FSA cannot count. Regular languages and their associated machines deal handily with zero, one, and infinity, but they cannot cope with the numbers in between.

The regular expression a^*b^* describes strings with any number of a's followed by any number of b's. Suppose there is some need for a language formed on the same model but with the additional constraint that the number of a's must equal the number of b's. The language can be described by the (nonregular) expression a^nb^n, where the superscript n signifies not exponentiation but concatenation n times.

If there is some limit on the value of n, this language can be recognized by an FSA with $2n$ states. (The states are arranged like railroad tracks with crossties; each a moves the machine one step out along one rail and each b moves it one step back along the other rail.) The situ-

A parity machine



Simple finite-state automaton recognizes the regular language made up of all strings of 1's and 0's with an odd number of 1's. A 0 input leaves the state of the machine unchanged, whereas each 1 causes it to toggle between states. The double circle for the Odd state signifies an accepting, or final, state.

Figure 2.

ation becomes more interesting when all limits on n are eliminated. No machine with a finite number of states can recognize this language.

A language made up of matched strings of a's and b's may seem artificial and unlikely to come up in practice. If a and b stand for "(" and ")," however, or for "begin" and "end," the utility of such constructions becomes apparent. They are commonplace in programming languages, and they appear in natural languages as well.

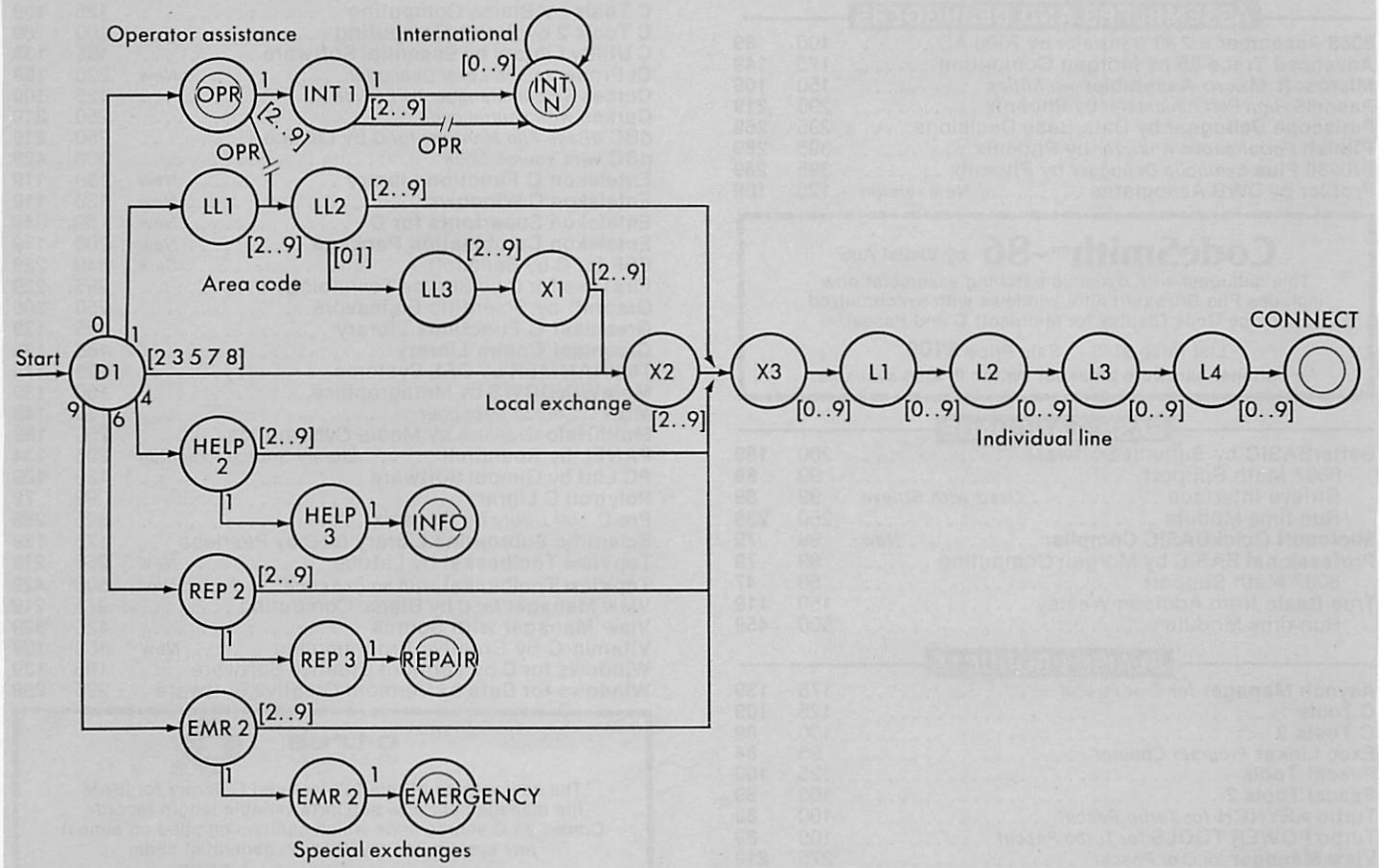
All balanced strings of parentheses (and only balanced strings) can be generated by the grammar:

S::=() | (S) | ()S

Note that this rule yields not only strings of the form "((()))" but also more complex nesting patterns such as "()()" and "(()())." On the other hand, it does not merely accept any string with equal numbers of left and right parentheses; "))((" and the like are excluded.

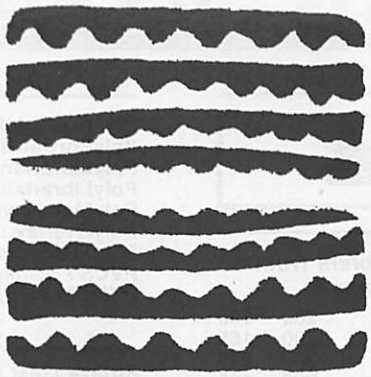
This result is achieved only at the cost of breaking the rules laid down for a regular grammar. In these productions the right-hand side does not consist of a single terminal or of a terminal followed by a nonterminal. Any string of symbols can appear on the right-hand side of a produc-

An FSA for telephone numbers



Valid telephone numbers can be recognized by a finite-state machine and therefore constitute a regular language. The machine passes through different sets of states for long-distance calls (beginning 1), operator-assisted calls (beginning 0), calls for directory assistance, repair service or emergencies (411, 611 and 911), and local calls. All area codes must have a middle digit of 0 or 1 and local exchanges cannot have a 0 or a 1 in their first two digits. Finite-state machines have had an important role in the development of switching theory for the telephone system and other applications.

Figure 3.



tion. There is one constraint still being observed, however: the left-hand side consists of a single symbol.

A grammar constructed according to these relaxed strictures is called a context-free grammar. The name reflects the isolation of the symbol on the left-hand side of each production. Because a symbol must appear alone there, the strings it generates cannot depend on the context in which it is found; a given nonterminal must always yield the same productions no matter what symbols surround it.

The machine that recognizes a context-free language is a pushdown automaton, or stack machine. It consists of an FSA with the addition of a potentially infinite pushdown stack, where items may be stored one on top of the other. The FSA has access only to the symbol at the top of the stack. It is easy to see how the stack can be used in recognizing a balanced expression. Each time a left parenthesis is encountered, it is pushed onto the stack. For each right parenthesis, one symbol is popped off the stack. A string is accepted if the stack is emptied just as the last input symbol is read.

Context-free languages form the second level in the Chomsky hierarchy, and they are by far the most important class for the construction of programming languages. The phrase-structure part of a natural language grammar is also generally stated in context-free format. There are a number of interesting subdivisions of the class, which I shall discuss in Part III of this series. For now, we must continue our climb up the tower.

The top of the heap

A context-free language can count two things, such as left and right parentheses, but what if it is necessary to keep track of three? Suppose the language to be recognized is $a^n b^n c^n$, where n can be any positive integer? No context-free language can accomplish this task. It is hard to see why in terms of the constraints imposed on a context-free grammar, but the reason becomes clear when the operation of a pushdown automaton is considered. If the *a*'s are pushed onto the stack and then popped off as the *b*'s are read, no record of the number of *a*'s and *b*'s remains when the *c*'s are counted.

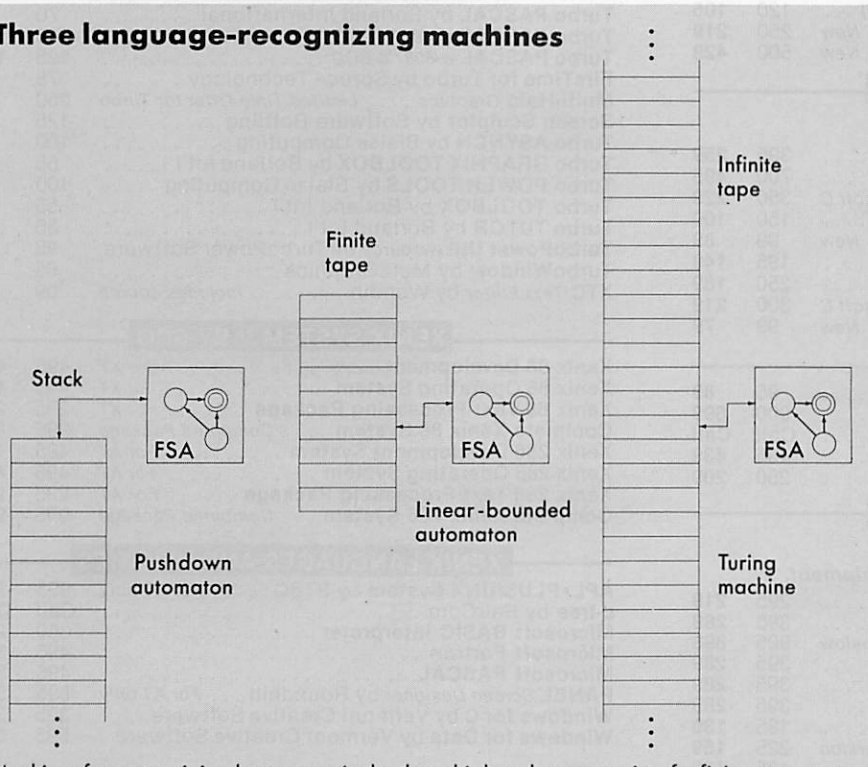
The machine needed to count three items is a linear-bounded automaton. It is an FSA augmented by a tape on which symbols can be written and then reviewed in any sequence. The machine has access to the entire tape at all times. The automaton is said to be linear-bounded because the maximum length of tape needed is a linear function of the length of the input. In other words the length of the tape is proportional to the length of the input.

A linear-bounded automaton might recognize $a^n b^n c^n$ by writing down all the *a*'s and *b*'s as they are received and then erasing one *a* and one *b* for each *c* found in the input. The sentence is accepted if the tape becomes blank just as the last symbol is read. This procedure can obviously be extended to the counting of more than three items.

There are other constructions recognized by a linear-bounded automaton but not by any lesser machine. For example, a pushdown automaton can check for symmetry in strings of the form *abcXcba*, but a linear-bounded automaton is needed to ensure matching in a pattern such as *abcXabc*. In the former case, where the second part of the string is reversed, the symbol needed in each stage of the matching process is available at the top of the stack. In the latter case a stack machine would bury the first *a* under the *b* and the *c*, and so it would be inaccessible when the second *a* was encountered. A linear-bounded automaton, on the other hand, can travel back over its tape to check off the symbols in any order.

The language recognized by a linear-bounded automaton is called a context-sensitive language. It is distinguished

Three language-recognizing machines



Machines for recognizing languages in the three higher classes consist of a finite-state automaton augmented by auxiliary storage. The pushdown automaton, which recognizes a context-free language, has a stack that allows access only to the top item. The linear-bounded automaton recognizes a context-sensitive language by exploiting a storage tape whose length is proportional to the length of the input. A Turing machine, with an infinite tape, is needed to recognize the sentences of a recursively enumerable language.

Figure 4.

from a context-free language by the abandonment of the requirement that the left-hand side of a production have only one symbol. As a result the expansion of a nonterminal can depend on the symbols that surround it. A grammar might include rules like the following:

aXb ::= auvb
cXd ::= cwzd

Here *X* yields one string of symbols in the context *aXb* and a different string in the context *cXd*.

Just one important restriction remains on the form of the productions in a context-sensitive language. In any rule of the grammar no more than one symbol on the left-hand side can be expanded on the right-hand side.

Context-sensitive constructions are rare in practical languages, but they do turn up. They are mainly of the form *abcXabc*. An example in English is the sentence "The butcher, the baker, and the candlestick maker spoke Finnish, Hungarian, and Basque, respectively." The same problem arises in programming languages that require all variables to be declared before they are used or that require a procedure's actual parameters to match its formal parameters.

What happens if all constraints on the form of a production rule are cast aside? The result is a grammar of type 0, the class at the top of Chomsky's pyramid. Any string of symbols can appear on either the left-hand or the right-hand side of a production.

The major limitation of the context-sensitive languages is that although they can count, they cannot calculate. The tape of a linear-bounded automaton can grow only in proportion to the input string, but many calculations require an amount of space that increases according to some nonlinear function; for example, the tape might grow exponentially. A type 0 grammar can accommodate such growth because the associated recognizing machine is a Turing machine—an FSA that can write on and read from a tape of unlimited length.

A language defined by a type 0 grammar is said to be recursively enumerable. The sentences of the language can be listed, or enumerated, by a Turing machine. Many of the languages are intriguing novelties, although so far they have proved to be of no practical use whatever. A typical example is the language of all strings of a's whose length is a power of 2; in other cases the length might be a perfect square or a prime number. The patterns are deeply rooted in mathematics but remote from linguistic issues.

In these languages the grammar itself has a mathematical flavor: it reads like a computer program. The derivation of a sentence in the language a^{2^n} is shown in Figure 6. Markers bounce from end to end in a growing string of a's, then collide and annihilate one another. The working of the grammar can hardly be distinguished from the operation of the underlying Turing machine, scanning back and forth over its tape.

There are languages beyond the recursively enumerable ones, but there are no grammars beyond type 0 and no recognizing machines beyond the Turing machine. The ethereal languages that cannot be reached from the top of Chomsky's skyscraper correspond to the noncomputable functions, those whose demands for space or time grow so fast that even a Turing machine cannot keep up with them. The existence of such functions and languages is one of the more unsettling discoveries of 20th century mathematics. They are of no conceivable use. Even if the need for such a language arose, no machine could recognize it.

The view from the top

Looking back over the four classes of languages, a pattern of ascending and descending currents can be perceived. Climbing up from type 3 to type 0, grammars become progressively more elaborate as the constraints on rule forms are relaxed. Every rule in a regular grammar could also appear in a type 0 grammar, but the latter class can include many other rules in addition. Each grammar class is a proper subset of the one above it.

For languages the trend is in the opposite direction. The constraints on form become tighter as one proceeds from type 3 to type 0, and each class is a subset of the one below it. All the sentences of the recursively enumerable language a^{2^n} are also members of the simple regular language a^*, but a^* includes many strings that are not sentences of a^{2^n}. If a type 0 language is powerful, that is not because it can say *aaaa*; a language in any of the classes can do that. The type 0 language is singled out because it cannot say *aaa* or *aaaaa*.

Context-sensitive grammar

Derivation of a string with equal numbers of consecutive a's, b's and c's is controlled by a context-sensitive grammar. The first two rules are the crucial ones. Rule 1 generates all strings with equal numbers of a's, X's and Y's, then Rule 2 rearranges the X's and Y's. The remaining rules merely substitute b's for X's and c's for Y's.

Grammar for $a^nb^nc^n$

1. S ::= aXY | aSXY
2. YX ::= XY
3. aX ::= ab
4. bX ::= bb
5. bY ::= bc
6. cY ::= cc

Derivation of aabbcc

S	
↓ Rule 1	
aSXY	
↓ Rule 1	
aaXYXY	
↓ Rule 2	
aaXXYY	
↓ Rule 3	
aabXYY	
↓ Rule 4	
aabbYY	
↓ Rule 5	
aabbcY	
↓ Rule 6	
aabbcc	

Figure 5.

Type 0 grammar

Grammar for generating sentences whose length is an integral power of 2 functions as a Turing-machine program. The symbols U and V mark the end of the string. X marches through the string from left to right, doubling the number of a's on each pass. Y and Z progress from right to left, then either start another cycle or terminate the process.

Grammar for a^{2^n}

1. S ::= UXaV
2. Xa ::= aaX
3. XV ::= YV | Z
4. aY ::= Ya
5. UY ::= UX
6. aZ ::= Za
7. UZ ::= ϵ

Derivation of aaaa

S	
UXaV	Rule 1
UaaXV	Rule 2
UaaYV	Rule 3
UaYaV	Rule 4
UYaaV	Rule 4
UXaaV	Rule 5
UaaXaV	Rule 2
UaaaaXV	Rule 2
UaaaaZ	Rule 3
UaaaZa	Rule 6
UaaZaa	Rule 6
UaZaaa	Rule 6
UZaaaa	Rule 6
aaaa	Rule 7

Figure 6.

Where do "real" languages fall in the Chomsky hierarchy? For natural languages the best answer seems to be "nowhere at all." Much of the observed syntax can be described by a context-free grammar, but most natural-language grammars also rely on a transformational component that is not based on production rules and so has no place in the hierarchy.

Programming languages are not much easier to classify. They are scattered in pieces over three of the four levels. Typically the fundamental lexical units, called tokens, form a regular language. They are entities such as numbers and variable names that can be recognized by a grammar that does no counting or calculation. The lexical scanner, the part of a compiler that breaks a continuous input stream into discrete tokens, is generally constructed as a finite-state automaton.

The bulk of a modern programming language is defined by a context-free grammar. All of the rules for handling arithmetic and logical expressions can be given in this format. A grammar with only context-free productions can also ensure that parentheses are balanced, that there is an "end" for every "begin," a "then" for every "if," and so on. A violation of these rules can therefore be detected early in the compilation of a program, during the checking of its syntax, and before any semantic interpretation begins. The software component that does the syntactic analysis is the parser; it can be modeled on a pushdown automaton.

Few programming languages, however, are entirely context-free. In Pascal and C every invocation of a function must include actual parameters that match (in number and type) the formal parameters given in the function declaration. No context-free grammar can ensure such matching. Thus it would seem that context-sensitive productions are needed in a grammar for Pascal or C.

In practice, that is not how it's done. The formal grammar of the language is given in context-free form, and features that cannot be made to fit the mold are left unspecified. The rule for a function declaration states merely that it can have a parameter list, which is defined in turn as any sequence of parameters and types. Checking for consistency between formal and actual parameters is done not by the parser but by an independent module of the compiler.

It is in the confrontation with "real" languages that both the strengths and the limitations of formal linguistic theory become most apparent. The guidance of a formal grammar is all but essential in the creation of a language, and yet it is not sufficient. The practicalities of language and compiler design will be the theme of Part III in this series.

References

Chomsky, Noam. "Three Models for the Description of Language." *IRE Transactions on Information Theory.* vol. 2, no. 3 (1956): 113-124. "On Certain Formal Properties of Grammars." *Information and Control.* vol. 2, no. 2 (1959): 137-167. [The two papers in which the hierarchy of language classes was first proposed.]

Hopcroft, John E., and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Reading, Mass.: Addison-Wesley Publishing Co. 1979. [A text that gives proofs for many of the assertions I have made but failed to defend.]

Minsky, Marvin L. *Computation: Finite and Infinite Machines.* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1967. [An exceptionally clear introduction, rigorous but never tedious.]

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.*

Artwork: Anne Doering