COMPUTATIONAL LINGUISTICS

Computational linguistics is concerned with the computer-based representation, processing, and discovery of information associated with human languages. This information encompasses such aspects as (1) the sounds used in words; (2) the structure of words and their formation from prefixes, suffixes, and other word elements; (3) the structure of phrases, sentences, and texts; (4) the meanings of such linguistic entities as words, phrases, and sentences; and (5) the use of language in context. These five aspects of linguistic information are related to linguistic research in the areas of (1) phonology, (2) morphology, (3) syntax, (4) semantics, and (5) pragmatics. The processing of this linguistic information typically involves analysis tasks, which use human language as input, and generation tasks, which produce human language as their output. This ties in with the closely related area known as natural language processing. While natural language processing can be viewed as the application of computational techniques to human language in general, computational linguistics is more concerned with the computational aspects of linguistic information. From this perspective, *applications* of computational linguistics would intersect with natural language processing.

Applications of computational linguistics are seen in such tasks as (1) machine translation, (2) grammar or style checkers, and (3) natural language interfaces to machines. Machine translation is concerned with the use of computers to translate from one human language to another (1). The degree of human intervention in this process can vary, so that one can have (1) fully automatic machine translation of a text, in which users do not require any knowledge to assist in the translation process; (2) human-assisted machine translation, in which user knowledge is applied to help with the translation; or (3) machine-assisted human translation, in which the computer provides tools (for example, on-line dictionaries, possible translations of specific phrases) to aid the human translator. Grammar and style checkers can be viewed as the next step in a sequence pioneered by spelling checkers. Documents can be processed so that unusual constructions can be flagged either interactively while they are being composed or off-line after they have been produced. For instance, the user can be notified if a sentence is missing a verb or if the subject and main verb of the sentence are inconsistent in their form. This requires a system that incorporates linguistic information concerning the structure of a language. Natural language can also be used to access the large amounts of information contained in computer databases and to instruct the computer how to perform complex tasks. Natural language is extremely attractive as input and output modes for a computer since it allows humans to communicate in a more natural way with the computer (much as they would with another human). It is also attractive for users who do not want to learn specialized artificial languages for communicating with machines; for example, natural language can be used in place of database query languages (2). Natural language gives users potentially enormous freedom of expression and can offer kinds of interaction different from graphic user interfaces. Some interfaces make use of speech input and output (3). Typically, the systems need a short training period for the user so that the system can adapt and optimize its performance. Sometimes, training is not possible so the system must rely on a user-independent strategy.

It is frequently possible to use the same processing techniques regardless of the choice of human language being processed. The amount of linguistic information associated with a language is vast, and it is a formidable task to isolate and encode the information. However, it is possible to create, automatically or semiautomatically, many linguistic data structures that can subsequently be used in various applications, as discussed in Ref. 4. These tasks for the discovery of linguistic information typically make use of collections of examples, existing resources such as text corpora (containing selected text from books, articles, etc.) (5), or even human-designed dictionaries.

NATURAL LANGUAGE

Ambiguity

Natural languages such as English are significantly different from artificial languages such as programming languages. Words used in natural languages are frequently highly ambiguous. For instance, the meaning for "down" includes (1) a direction, (2) soft feathers such as those used in quilts and pillows, (3) an emotional state of feeling down and depressed, or (4) an action when a fighter can "down an opponent with a single blow." Ambiguity is also present in the syntax or structure of natural language, since a given expression can frequently have numerous alternative structures associated with it. A sentence like "I like water in the spring" is highly ambiguous not only due to the different meanings of the words, but also because the phrase "in the spring" can be modifying water, as in water that comes from a cold spring, or it can be modifying the action that is taking place, as in response to the question "When do you like water?" Ambiguity is also introduced by such words and phrases as pronouns (e.g., it). Pronoun (pronominal) ambiguity can be seen in the sentence "I like water in the spring when it is cold," in which "it" can correspond to "water," to "spring," or could even be used in a nonreferring sense, as part of the phrase "it is." A grammar that attempts to capture all the information relevant to examples like these will contain a great deal of ambiguity, so it is not unusual for a sophisticated grammar to have hundreds and sometimes thousands of possible analyses for some sentences.

Modeling Natural Language

Linguistic information needs to be identified and then represented in a manner that is useful to both a human and a computer. There are a variety of formats that can be used to represent linguistic information. Some formats are well suited for the representation of specific kinds of information. For example, formats based on first-order logic can be used to describe information relating to linguistic meaning or semantics, while grammars are typically used to represent structural or syntactic information. A grammar contains rules that describe the set of sentences that make up a language. There are even some formats in which both syntactic and semantic-based information can be described. The format that is used to represent this information is determined by the "formalism." Formalisms designed for dealing with the grammar of a language are referred to as "grammar formalisms." Formalisms differ depending on factors such as (1) what set of languages they can represent, (2) how efficiently they can be processed by a machine, (3) how they actually represent linguistic information, and (4) how easily they can be used by a human. The first two factors are reflected in the "power" of the formalism. The choice of formalism frequently depends on the type of linguistic phenomena that are being taken into account and the specific task at hand.

Depending on the grammar formalism used to describe linguistic information, there are different computational models for processing the information. Some of these models allow for very efficient processing, while others can require extremely intensive computation. If a formalism is used that is extremely powerful in discriminating which sentences are and are not in a language, then the amount of computation needed to recognize a sentence can be very great. For formalisms that are not discriminating, the amount of computation needed to recognize a sentence can be relatively small. It thus comes as no surprise that when powerful formalisms are used, it is sometimes necessary to take shortcuts when the grammar is used for specific applications (taking merely an approximation of the set of sentences described by the entire grammar); grammars can be compiled or translated from one form to another.

The Chomsky hierarchy is frequently used to illustrate the relationship between the power of different grammar formalisms and the relationship between the languages associated with the formalisms (6). The relationship of the Chomsky hierarchy to natural languages has also been examined in detail (7). At the bottom of the hierarchy is the set of regular grammars (also known as type 3 grammars), which correspond to the set of regular languages. Regular grammars are the least powerful grammar formalism in the hierarchy, and the set of regular languages represents the simplest languages. Regular languages can be recognized and generated by a finite-state machine in an amount of time proportional to the length of the sentence. Context-free grammars, also known as type 2 grammars, are a proper superset of regular grammars. The set of regular languages is thus a proper subset of the set of context-free languages. The automaton corresponding to this class is the pushdown automaton, which is just a finite-state machine having a single stack as a storage device. In the worst case, a sentence from a context-free grammar can be parsed in an amount of time proportional to the cube of the length of the sentence. Subcubic algorithms for recognizing, as opposed to parsing, sentences have also been developed. Context-sensitive grammars (type 1) are, for all practical purposes, a proper superset of context-free grammars, and their corresponding languages are also in a superset relationship. The automaton associated with these grammars has an "infinite tape" as its storage device, and there are some restrictions on the operations that it may perform. There are no efficient algorithms for recognizing or parsing context-sensitive languages in general, but given a sequence of words, it is always possible to say whether or not it is "grammatical" (whether or not it is part of the set of sentences that make up the language). Later in this article we describe some "mildly context-sensitive" formalisms that do have polynomial time algorithms for sentence recognition or parsing. Unrestricted rewrite grammars (type 0) are at the top of the Chomsky hierarchy. Every recursively enumerable set is generated by some type 0 grammar, and every type 0 grammar generates a recursively enumerable language. Thus, when given a sequence of words, it is always possible to say whether it is a grammatical sentence according to a type 0 grammar, but it is not necessarily possible to say that it is ungrammatical. It is important to note that linguistic formalisms are not the only means for modeling natural language. Models based on acceptability or based on comprehensive sets of examples are also possible, as discussed in statistical language modeling (5).

Use of Procedural Information

The distinction between the representation of linguistic information and the processors of that information is really just the familiar distinction between data and processes that is important in all areas of computer science. With this distinction, linguistic processes that work with one language can easily be adapted to work for other languages using similar data structures. In addition, the task of maintaining an existing computational linguistic application is simplified. The clear separation of linguistic information from procedural information is characteristic of "declarative formalisms." Woods' augmented transition networks (ATN) are one example of a more procedural formalism that allows actual LISP programming language code to be mixed with the linguistic information (8). Similarly, definite clause grammars allow Prolog programming language code to be included in grammar rules (9). While such approaches may make the linguistic knowledge less apparent, they frequently can lead to improved performance in applied systems.

GRAMMAR FORMALISMS

There are a large number of grammar formalisms that have been proposed and used in computational linguistics. While the differences between formalisms may range from extreme to insignificant, a number of concepts can be viewed as common to many formalisms. Formalisms tend to have primitives for describing the basic building blocks of language, along with rules for stating how complex structures are built from more primitive structures. In our discussion of grammars, we will be focusing on syntactic or structural aspects of language. Although semantic information (6) can be incorporated into a grammar, or into a natural language processor, it is not of primary concern to us here. Similarly, we will not be concerned with morphological (10) or phonological information, though these kinds of information can play an important role in speech recognition (3). We will also not be looking at the mathematical aspects of formalisms or languages in any great detail (6,11). We will now consider a selection of the more widely known grammar formalisms.

Context-Free Grammars

Context-free grammars (CFG) have played an influential role in computational linguistics. Aspects of CFGs are reflected in different grammar formalisms. Indeed, many grammar formalisms tout a "context-free backbone," having rules patterned after context-free rules but augmented with additional information. There are various parsing algorithms that have been designed to process CFGs efficiently and formalisms having a context-free backbone. CFGs are closely related to Backus–Naur form (BNF) specifications (12), which are frequently used to describe formally the syntax of programming languages and which are also used to define recursive data structures in computing science applications.

Components of a Context-Free Grammar. Let us now consider the components of a CFG. A CFG consists of (1) a set of terminal symbols corresponding to the words or tokens of the language (such as *mother* or *like*); (2) a set of nonterminal symbols corresponding to constituents or classes of constituents found in a language (such a *sentence* or *relative clause*); (3) a set of grammar rules of the form "$A \rightarrow \alpha$," where A is a nonterminal symbol and α is a sequence of zero or more terminal and nonterminal symbols; and (4) a designated nonterminal symbol known as the start symbol, a constituent or class for all valid/grammatical sentences of the language. For cases where α is the empty sequence, a special symbol like λ or ϵ is often used.

Context-Free Grammar Example. Equations (1) to (10) introduce a collection of grammar rules for approximating a small subset of English, where the terminal symbols are shown in italics. Nonterminal symbols that introduce a terminal symbol are often called preterminal symbols. So the nonterminals in Eqs. (7) to (10) are considered to be preterminals. Equations (9) and (10) also illustrate the use of disjunction in rules, which is represented with a vertical bar. In each of these two cases, a single disjunctive rule could be replaced by two alternative rules without disjunction, one for each alternative in the disjunction.

$$\text{sentence} \rightarrow \text{noun\_phrase, verb\_phrase} \tag{1}$$

$$\text{verb\_phrase} \rightarrow \text{verb} \tag{2}$$

$$\text{verb\_phrase} \rightarrow \text{verb, noun\_phrase} \tag{3}$$

$$\text{relative\_clause} \rightarrow \text{relative\_pronoun, sentence} \tag{4}$$

$$\text{noun\_phrase} \rightarrow \text{noun\_phrase, relative\_clause} \tag{5}$$

$$\text{noun\_phrase} \rightarrow \text{determiner, common\_noun} \tag{6}$$

$$\text{determiner} \rightarrow \textit{the} \tag{7}$$

$$\text{relative\_pronoun} \rightarrow \textit{that} \tag{8}$$

$$\text{common\_noun} \rightarrow \textit{cat}|\textit{dog} \tag{9}$$

$$\text{verb} \rightarrow \textit{likes}|\textit{hates} \tag{10}$$

The language of the grammar is the set of sequences of terminal symbols (words) that can be generated according to the rules of the grammar, given the start symbol. Corresponding to grammatical sentences are parse trees, or derivation trees, which indicate which rules are responsible for the sentence. A sentence that has more than one parse or derivation tree is said to be ambiguous. The language of the preceding CFG allows sequences of words that constitute sentences like *The dog that the cat likes hates the cat* or *The dog hates the cat* or even *The dog that the cat that the dog hates likes hates the cat,* but excludes sentences like *The dog the cat hates.* Observe that this grammar is only a rough approximation since it would also allow unacceptable sentences like *The dog that the cat hates the dog hates the cat.* Figure 1 shows a tree for the sentence *The dog that the cat likes hates the cat*.

Observe that some of the aforementioned grammatical sentences seem very unnatural or difficult to understand. However, a traditional linguistic analysis would deem some of them to be valid sentences of the English language. This raises the issue of *competence* versus *performance,* which was discussed by Chomsky (13). When using language, we may employ and understand expressions that we would consider to violate the rules of the language. Such sentences reflect performance aspects of language use. In contrast, there are sentences that, when analyzed, would conform to the accepted rules of the language yet would probably not be employed or understood by a typical native speaker. This competence aspect of language use is exemplified by the 13-word sentence *The dog that the cat that the dog hates likes hates the cat.* Indeed, current research in natural language processing is concentrating more on the performance aspect during the development of robust natural language processing systems with broad coverage.

Categorial Grammars

Categorial grammars (CGs) also have a long history associated with them, as discussed in Ref. 14, and have the same power as CFGs. The primary difference from CFGs lies in the categories that are associated with the terminal symbols (words) of the grammar. While CFGs have atomic nonterminal symbols, CGs allow structured category names that are recursively composed of other category names. We can define a CG to consist of (1) a set of atomic categories; (2) a set of complex categories of the form α/β or $\alpha\backslash\beta$, where each of α and β are themselves categories (either atomic or complex); (3) two rules, one for forward functional application, as shown in Eq. (11), and one for backward functional application, as shown in Eq. (12); and (4) a start category.

$$\alpha \rightarrow \alpha/\beta, \beta \tag{11}$$

$$\alpha \rightarrow \beta, \alpha\backslash\beta \tag{12}$$

CGs are frequently viewed as a lexical formalism because of the heavy reliance on the information associated with the different lexical items (words) and the minimal information associated with the rules—as presented previously, CGs have only two rules. Each of these two rules can also be viewed as schema describing a large set of rules that have specific category names rather than variables like α or β. Intuitively, when a word or a constituent has a category name of the form α/β, it can be viewed as a function that looks for its argument β on its right in order to become an α, as reflected in the grammar rule from Eq. (11). Similarly, a constituent with category name $\alpha\backslash\beta$ is a function looking for a β on its left, Eq.

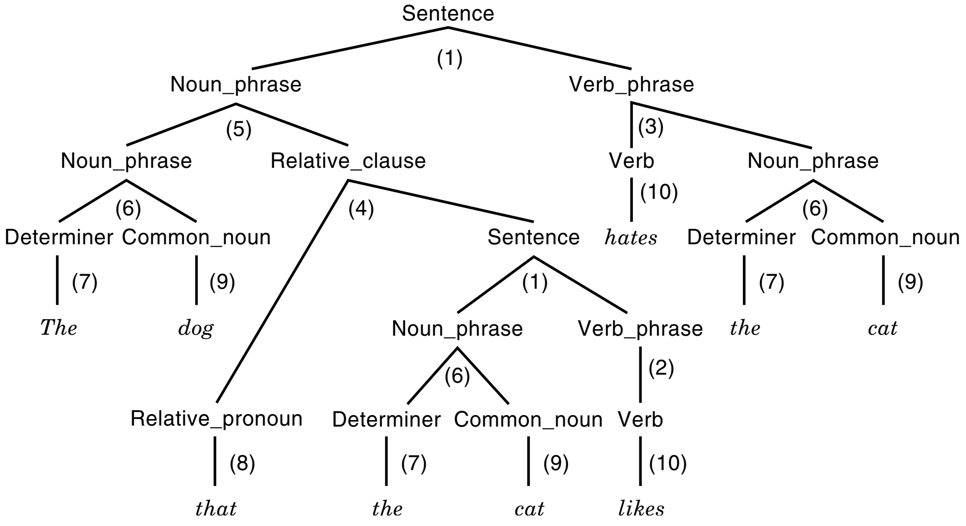

Figure 1. Context-free grammar parse tree of the sentence *The dog that the cat likes hates the cat.* The terminal symbols (leaves) of the tree correspond to words, while the nonterminal symbols (internal nodes) correspond to classes or constituents. The branches correspond to grammar rules; the numbers in parentheses reference the equation numbers of the grammar rules used.

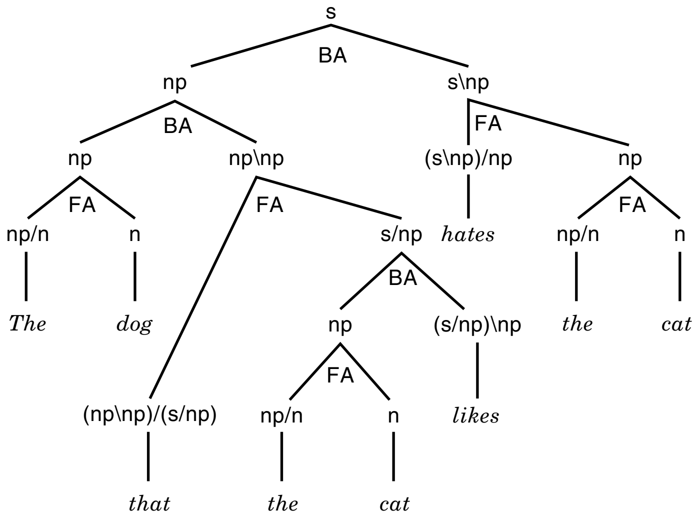

Figure 2. Categorial grammar parse tree of the sentence *The dog that the cat likes hates the cat.* Branches are labeled with the names of the grammar rules used, either forward functional application (FA) or backward functional application (BA).

(12). Alternatively, either an α/β or an $\alpha\backslash\beta$ can be viewed as an α that is “missing” a β.

Categorial Grammar Example. The following example shows that essentially the same structure can be obtained for the sentence *The dog that the cat likes hates the cat,* as was obtained using the CFG from Eqs. (1) to (10). Equations (13) to (20) show the categories for the different words, where s (sentence), np (noun_phrase), and n (common_noun) are introduced as the only atomic categories. Figure 2 shows the corresponding parse tree, where the branches are labeled with the name of the grammar rule used, either FA for forward functional application or BA for backward functional application.

$$\textit{cat}\colon \mathrm{n} \tag{13}$$

$$\textit{dog}\colon \mathrm{n} \tag{14}$$

$$\textit{the}\colon \mathrm{np/n} \tag{15}$$

$$\textit{that}\colon (\mathrm{np}\backslash\mathrm{np})/(\mathrm{s/np}) \tag{16}$$

$$\textit{likes}\colon (\mathrm{s}\backslash\mathrm{np})/\mathrm{np} \tag{17}$$

$$\textit{likes}\colon (\mathrm{s/np})\backslash\mathrm{np} \tag{18}$$

$$\textit{hates}\colon (\mathrm{s}\backslash\mathrm{np/np} \tag{19}$$

$$\textit{hates}\colon (\mathrm{s/np})\backslash\mathrm{np} \tag{20}$$

Observe that the lexical entries for *hates* and *likes* in Eqs. (17) to (20) are ambiguous—each word has a choice of two possible category assignments. This ambiguity means that full sentences like *The dog the cat hates* are allowed that were not allowed by the CFG given earlier. In contrast, the CG does not allow sentences like *The dog that the cat hates the dog hates the cat,* that were allowed by the CFG. The grammar will also allow two analyses for a simple sentence like *The dog hates the cat.* This last ambiguity is not linguistically significant, so it is deemed to be a “spurious ambiguity.” Spurious ambiguities are not limited to just CGs but are widespread in many large grammars.

Grammar and Formalism Equivalence. Our comparison of CG and CFG analyses of the same sentence leads us to another important issue relating grammar formalisms and grammars: Two grammars are strongly equivalent if they generate the same structures, and they are weakly equivalent if they generate the same set of sentences. Two formalisms are thus strongly equivalent if every grammar in one formalism has a strongly equivalent grammar in the other formalism. The formalisms would be weakly equivalent if every grammar in one formalism had a counterpart in the other formalism that would generate the same language (sets of sentences). While this strong/weak distinction may be important for some discussions, it need not concern us here. We can restrict our discussion to weak equivalence when discussing the relative power of different grammar formalisms.

Combinatory Categorial Grammars

While traditional CG incorporates only two grammar rules, combinatory categorial grammar (CCG) allows a greater variety of grammar rules (15). The motivation for exactly which rules are introduced is based on specific “combinators.” By introducing additional grammar rules, some forms of lexical ambiguity [see, for instance, Eqs. (17) to (20)] can be decreased. However, this often comes at the cost of increased structural ambiguity and can lead to spurious ambiguities, where the ambiguity present in the grammar is merely an artifact of the grammar and does not correspond to true syntactic ambiguity in the natural language being modeled. CCGs also allow the description of languages that cannot be described by CFGs. Thus, they have more than context-free power—they belong to the class of mildly context-sensitive grammar formalisms (16). In addition to the forward and backward functional application rules, the rules that are usually associated with CCG include functional composition, type raising, and functional substitution, which are described in turn next.

Functional Composition. Functional composition rules allow the combination of two complex categories. There are forward and backward versions of functional composition, mirroring the forward and backward versions of functional application.

$$\alpha/\gamma \rightarrow \alpha/\beta,\ \beta/\gamma \tag{21}$$

$$\alpha\backslash\gamma \rightarrow \beta\backslash\gamma,\ \alpha\backslash\beta \tag{22}$$

There are also variations on these rules depending on the direction associated with the slash in the various constituents. The notion underlying all of these rules is the same: There is one function that is looking for an argument β to return an α, and there is another function that is looking for an argument γ to produce a β. These two functions can be composed to produce a function that is looking for a γ to produce an α. Sometimes a direction independent version of a rule like functional composition is given, as shown in Eq. (23), where a vertical bar is used as a slash with an unspecified direction.

$$\alpha|\gamma \rightarrow \alpha|\beta,\ \beta|\gamma \tag{23}$$

Type Raising. Type raising is a unary rule that allows one to change a category in a restricted manner. Until now, we have seen only binary rules in categorial grammar. Type raising effectively allows a constituent of some category α that is an argument of a function of some category φ to change its category and become a function that takes an φ as its argument. This results in rules for forward and backward type raising.

$$\varphi/(\varphi\backslash\alpha) \rightarrow \alpha \tag{24}$$

$$\varphi\backslash(\varphi/\alpha) \rightarrow \alpha \tag{25}$$

Restrictions on these rules usually take the form of different categories that are allowed for α and φ. Notice that without any restrictions on a type-raising rule, it could act like a schema corresponding to potentially an infinite number of rules, where the variables α and φ are replaced with actual categories.

Functional Substitution. The functional substitution rules allow two functions looking for the same category of argument to be combined into a single function still looking for that argument. As with functional composition, there are variations on this rule depending on the direction of the slash in the categories of the constituents. Equation (26) shows the rule without the direction information.

$$\alpha|\gamma \rightarrow (\alpha|\beta)|\gamma, \beta|\gamma \tag{26}$$

The first function is "looking for" a γ argument and a β argument in order to produce an α. The second function needs a γ argument to become a β. Thus, the rule allows the first function to combine with the second function, even though the γ argument has not yet been encountered. The resulting function then still needs to find the γ.

Combinatory Categorial Grammar Example. Let us now consider a CCG that can be used to provide an analysis of the same sentence considered in previous sections, *The dog that the cat likes hates the cat.* Our grammar can use the same lexical entries that were introduced in Eqs. (13) to (16), supplemented by unambiguous lexical entries for *hates* and *likes.*

$$\textit{likes}: (\mathrm{s}\backslash\mathrm{np})/\mathrm{np} \tag{27}$$

$$\textit{hates}: (\mathrm{s}\backslash\mathrm{np})/\mathrm{np} \tag{28}$$

Figure 3 illustrates how the type-raising (TR) rule from Eq. (24), used in conjunction with the functional composition (FC) rule from Eq. (21), results in an analysis of the sentence. Viewing the tree from the bottom up, observe how application of type raising to the constituent *the cat,* which follows the relative clause, allows it to be promoted to a function that can then be combined with *likes* according to functional composition from Eq. (21). So the same effect that was obtained in traditional CG is achieved without lexical ambiguity, but at the cost of additional grammar rules. However, spurious ambiguity is introduced if we consider that type raising could be applied to the subject of the sentence, *The dog that the cat likes,* and then the type-raised subject could then instead take *hates the cat* as its argument.

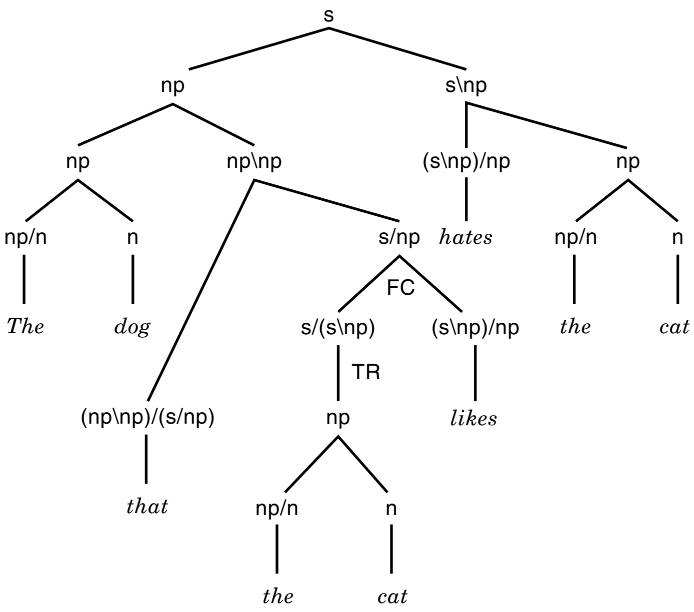

Figure 3. Combinatory categorial grammar parse tree of the sentence *The dog that the cat likes hates the cat.* The rules used for the branches are the same as in Fig. 2 except when explicitly specified as type raising (TR) or functional composition (FC).

String Grammars

String grammars (12) incorporate CFGs but introduce additional mechanisms that result in a formalism more powerful than CFGs. Thus far, a nonterminal symbol in a grammar has been used to refer to a string of terminal symbols (words), and this string of terminal symbols has been defined recursively as the concatenation of the string of words associated with its immediate subconstituents. Returning briefly to Fig. 3, the nonterminal symbol s, for example, refers to the entire sentence, with np corresponding to *The dog that the cat likes* and s\np corresponding to *hates the cat.* String grammars use context-free rules (in BNF notation) to describe "linguistic strings," which are sequences of terminal and preterminal symbols. Linguistic strings are placed into different general classes depending on the role that the strings play in the language. There are two operations, adjunction and substitution, which allow linguistic strings of selected types to be combined, subject to a collection of constraints or restrictions. Restrictions are stated in a specially designed programming language; thus string grammars have a more procedural (less declarative) flavor than the other formalisms we have been discussing.

Example of String Operations. The ideas behind linguistic strings are illustrated in the following example. Assuming the CFGs introduced in Eqs. (1) to (10), Eq. (29), and Eq. (30) show linguistic strings for the sentence *The dog hates the cat* and for the phrase *that the cat likes,* respectively.

determiner noun verb determiner noun (29)

relative_pronoun determiner noun verb (30)

The string in Eq. (30) could be inserted into Eq. (29) using the adjunction operation to the right of the first noun, to result in a linguistic string, shown in Eq. (31), that would be associated with a sentence like *The dog that the cat likes hates the cat.*

determiner noun relative_pronoun determiner
noun verb verb determiner noun (31)

Head Grammars

With the introduction of head grammars (17), we see yet another variation on the traditional context-free style grammar rule that results in a grammar formalism having essentially the same power as CCG. Computational linguists do not tend to use head grammars, but this formalism has affected how string combination is viewed in a wide range of grammar formalisms. Head grammars influenced developments in head-driven phrase structure grammar (HPSG) (18).

Headed Strings. Rules in a head grammar describe how "headed strings" are combined rather than the traditional strings we have seen in CFGs, CGs, or CCGs. In this respect, they resemble string grammars. However, the characterization of head grammars is much more formal and declarative, with the formalism not relying on arbitrary restrictions that are stated in a programming language. A headed string is simply a string in which exactly one position in the string (one word) is designated to be the head. While traditional grammars combine strings simply using concatenation, head grammars also allow one string to be inserted into another string at its head position. Thus, we get a wrapping operation that is similar in effect to the adjunction operation from string grammars. Adopting Pollard's (17) notation of having the head preceded by a *, a head grammar rule could allow the headed string *The *dog hates the cat* to be combined with *that the *cat likes* to produce *The *dog that the cat likes hates the cat.*

Operations on Headed Strings. The grammar rules must specify how the head of the resulting string is determined from the heads of the constituent headed strings. So for a traditional concatenative grammar rule involving two constituents, there are two different possibilities: one in which the head of the left subconstituent becomes the head of the resulting constituent, and one in which the head of the right subconstituent is used. For a binary rule involving wrapping, the first constituent can either be wrapped around the second, or the second can be wrapped around the first. In addition, the wrapping position can either be before or after the head. This results in four possible wrapping relationships so far, and when we take into account that it is either the head of the first or second constituent that becomes the head of the resulting constituent, we have a total of eight different wrapping relationships. In the example in the previous paragraph, we saw the first constituent wrap around the second constituent at the position following the head of the first constituent, with the head of the first constituent becoming the head of the resulting constituent. Repeated application of this operation would result in multiple relative clauses modifying the head *dog.* A simple modification to the head grammar formalism results in normalized head grammars (19), which require fewer wrapping operations and are more closely related to CCG and tree adjoining grammar (16).

Tree Adjoining Grammars

Tree adjoining grammars (TAGs) (20) have trees, rather than symbols or strings of symbols, as their basic building blocks and introduce an adjunction operation for combining trees. Thus, they share some of the same underlying concepts found in string grammars, but the data structures are different and the properties of TAGs are much more well defined and understood. There is a wide range of formalisms that belong to the TAG family, but they all have the same underlying principles. Trees are defined as primitives, and there are two basic types of trees: (1) elementary trees, in which the leaves are terminal nodes, resembling the parse trees obtained according to a traditional CFG; and (2) auxiliary trees, in which the leaves contain one nonterminal node among the other terminal nodes and in which this single nonterminal node has the same name as the root node of the tree.

Adjunction Example. Figure 4 (left) shows an example of an elementary tree and Fig. 4 (right) an auxiliary tree corresponding to the sentence *The dog hates the cat* and to the phrase *that the cat likes,* respectively. The adjunction operation has the same effect as the adjunction operation of string grammars or the wrapping operation of head grammars. Adjunction allows the creation of a more complex tree by adjoining an auxiliary tree to a node in an elementary tree. When the auxiliary tree from Fig. 4 (right) is adjoined to the first noun_phrase node of the elementary tree from Fig. 4 (left), we obtain a tree identical to the one that was originally introduced in Fig. 1 for our full example sentence.

Restrictions on Adjunction. There are restrictions that may be placed on the adjunction operation, just as there were constraints on adjunction in string grammars. Nodes in trees may be designated as obligatory adjunction sites or as optional adjunction sites. In addition, each node may have restrictions concerning which auxiliary trees (if any) may be adjoined at that site. These restrictions on the adjunction operation differ considerably from those used in string grammars in that they do not require the power of a full programming language—the restrictions in TAG need only reference other trees.

Other Tree Operations. Adjunction is not a transformation in the sense traditionally used in transformational grammar (21). Tree transformations are known to be computationally demanding, since the introduction of transformations allows the description of a type 0 language. Much more efficient techniques have been developed for processing the adjunction operation. Adjunction can, however, be used to implement substitution, in which a nonterminal symbol present as a leaf on a tree is replaced by an entire tree. Some variations of TAG make explicit use of the substitution operation.

Linear Indexed Grammars. Gazdar's linear indexed grammars (LIGs) also belong to the class of mildly context-sensitive grammar formalisms (16), but they achieve this "greater than context-free power" by augmenting the structure of the nonterminal symbols used by the grammar rules (22). A (potentially infinite) sequence of indices is associated with each nonterminal symbol in the grammar. Aho introduced this generalization to create indexed grammars (23), which were

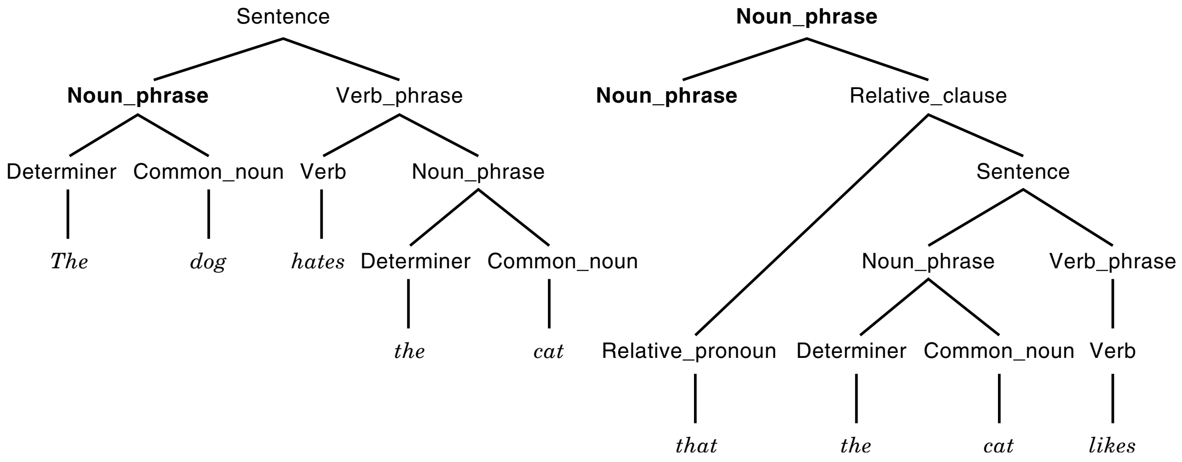

Figure 4. Elementary tree (left) and auxiliary tree (right) from a tree adjoining grammar. The intended adjunction site in the elementary tree is shown in bold. The root and foot of the auxiliary tree are also shown in bold. Adjunction occurs by effectively inserting the auxiliary tree in place of the node at the adjunction site in the elementary tree.

not linguistically motivated and which are, in fact, more powerful than CCGs, head grammars, or TAGs. Indexed grammar rules treat the sequences of indices associated with one symbol as a stack, and they allow (1) the stack to be copied, (2) an element to be pushed onto a stack, or (3) an element to be popped from a stack. By restricting the form of grammar rules used by the formalism, we obtain a formalism of more restricted power; we obtain LIGs, which have the same power as CCGs, head grammars, and TAGs. In LIGs, at most one symbol from the right-hand side of a grammar rule may have a nonempty stack.

Linear Indexed Grammar Example. The following example shows a grammar in which the indices stack is used to keep track of the missing noun phrases in a constituent. We start by assuming the same grammar rules as presented in Eqs. (6) to (10), where we assume that empty stacks are associated with each of the nonterminal symbols in the grammar. We then introduce the rules in Eqs. (32) to (34) for copying stacks, Eq. (35) for introducing an index into a stack (whenever a relative pronoun in encountered), and Eq. (36) for removing an index from a stack (whenever a noun_phrase is realized as the empty sequence λ).

$$\text{sentence}[..] \rightarrow \text{noun\_phrase}[], \text{verb\_phrase}[..] \tag{32}$$

$$\text{noun\_phrase}[..] \rightarrow \text{noun\_phrase}[], \text{relative\_clause}[..] \tag{33}$$

$$\text{verb\_phrase}[..] \rightarrow \text{verb}[], \text{noun\_phrase}[..] \tag{34}$$

$$\text{relative\_clause}[..] \rightarrow \text{relative\_pronoun}[], \text{sentence}[t, ..] \tag{35}$$

$$\text{noun\_phrase}[\text{t},..] \rightarrow \lambda \tag{36}$$

The resulting grammar takes into account the property of English that a sentence within a relative clause is effectively "missing" a noun phrase. Recall that the categories in categorial grammar also allowed us to keep track of missing constituents, and the use of a stack to keep track of missing constituents can also be seen in generalized phrase structure grammar (GPSG) (24) as well as HPSG. Figure 5 shows the parse tree for the sentence *The dog that the cat likes hates the cat.*

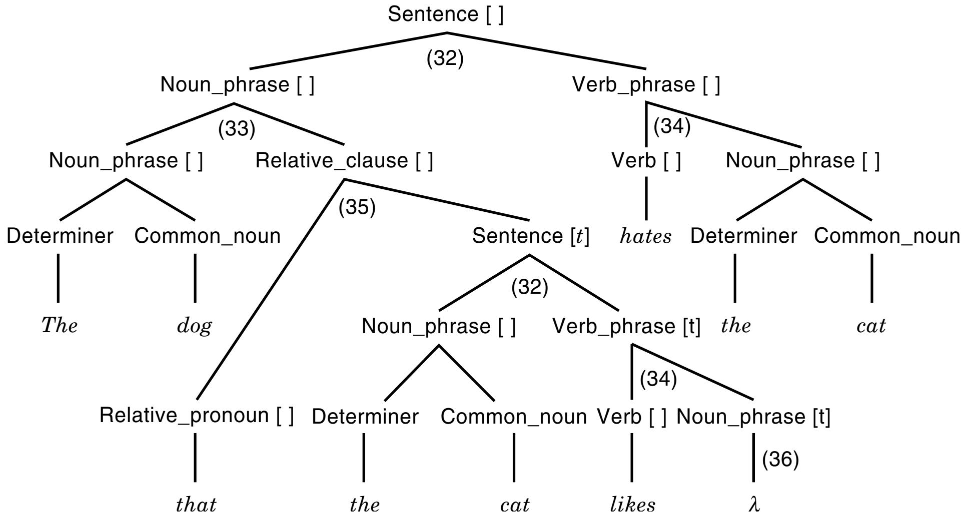

Figure 5. Linear indexed grammar parse tree for *The dog that the cat likes hates the cat.* The branches and nodes associated with the linear indexed grammar rules in Eqs. (32) to (36) are labeled. The empty stacks for the other nodes are not displayed. Equation (35) introduces an index *t* into the sentence stack, which is propagated by the grammar rules, allowing Eq. (36) to introduce the "empty" noun phrase.

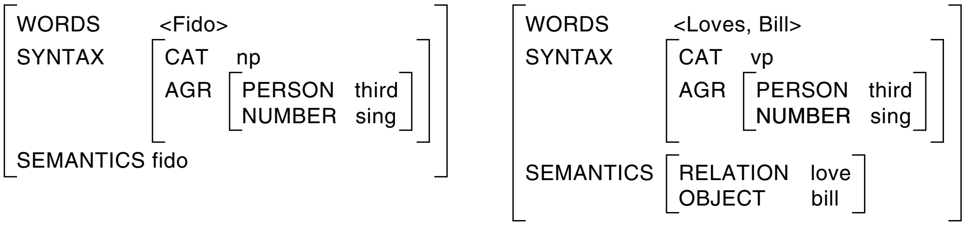

Figure 6. Feature structures associated with the term *Fido* (left) and the phrase *loves Bill* (right). The value of the WORDS feature is a list of words enclosed in angle brackets. The SYNTAX feature takes another feature structure as its value, having features for category (CAT) and syntactic agreement (AGR). AGR also has a feature structure as its value that states that each constituent is in the "third person" (as opposed to first or second person) and is singular (as opposed to plural). Each feature structure contains a very simplified SEMANTICS feature. For the np, we have introduced an atomic value corresponding to the entity, while for the vp we have introduced a nested feature structure that introduces a RELATION and an OBJECT of the relation.

Unification-Based Formalisms

Unification-based formalisms (25) allow more complex (and potentially recursive) structures called "feature structures" to be used as the basic building blocks that are subject to the operations associated with a formalism. In this respect, their building blocks are similar to those of categorial grammars and indexed grammars, but the structures can be much more complex. A feature structure is simply a set of attribute value pairs, where each feature (also known as an attribute) is an atomic "name" and where each value is either atomic or is itself another feature structure. There are several variations of feature structures and of the kinds of information that they can express. Some theories introduce the notion of typed feature structures, in which each feature structure possesses a "type" in addition to a set of attribute value pairs (26). Sometimes values are allowed to be sets, lists, and so forth. For example, Fig. 6 (left) shows the feature structure that might be associated with term *Fido* and Fig. 6 (right) the feature structure for *loves Bill.* One desirable aspect of feature structures is that they can contain more than just syntactic information, as illustrated in Fig. 6.

Often, theories make distinctions between feature structures themselves and a language for describing feature structures. For example, the feature structures themselves might not be allowed to contain variables, but descriptions of feature structures may be allowed to contain variables, disjunctions, and other operations, thus allowing a description to denote a whole set of feature structures.

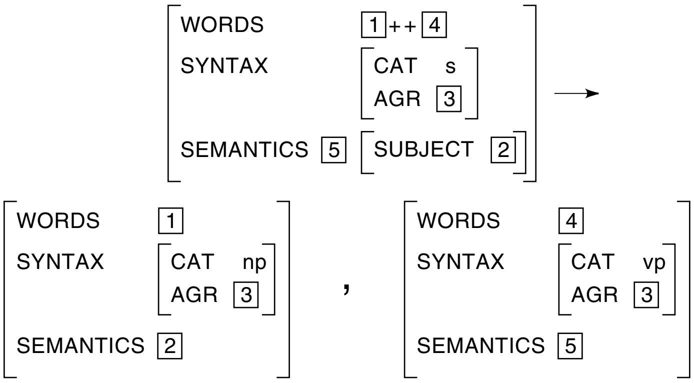

Figure 7. A grammar rule for combining feature structures. The presence of a number surrounded by a box acts like a pointer; any value referenced by one pointer is shared by other occurrences of the same pointer. The notation ++ is used to denote concatenation of lists.

Unification-Based Rules. The operations or rules of the grammar state how feature structures associated with different constructions are related to the feature structure associated with a more complex constituent. Rules in unification-based formalisms often resemble those in a CFG except that feature structures are used in place of the terminal and nonterminal symbols of a CFG. For example, Fig. 7 shows a grammar rule for combining noun phrases and verb phrases.

The notion of unification comes from the basic operation that is performed on feature structures through the grammar rules. Unification of two feature structures can informally be thought of as an operation that combines the information present in two feature structures, along with a requirement that the two feature structures not contain incompatible information. If unification is attempted on two feature structures that contain incompatible information, then the unification operation does not take place and is said to be undefined. For example, the feature structure in Fig. 6 (left) will unify with the first feature structure on the right-hand side of the grammar rule introduced in Fig. 7. The feature structure in Fig. 6 (right) will unify with the last feature structure from the grammar rule. An attempt to unify the feature structure in Fig. 6 (left) with the last one in the grammar rule would be undefined due to conflicting information. Figure 8 shows the re-

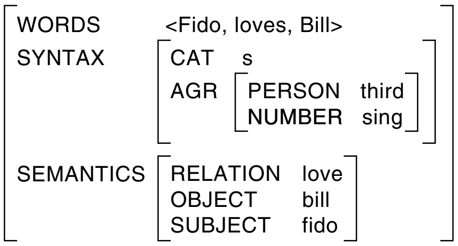

Figure 8. The feature structure resulting from application of the grammar rule to two feature structures. The SEMANTICS feature of the feature structure contains some information supplied by the first feature structure and some supplied by the second feature structure.

sult of using the rule from Fig. 7 to combine the feature structures from Fig. 6.

Power of Unification-Based Formalisms. Unification-based formalisms have much more than context-free power and clearly have much more than mildly context-sensitive grammars, since it would require only one feature in a feature structure to act as a stack and thus obtain an indexed grammar. In fact, unification-based grammars can express any recursively enumerable language since any type 0 grammar from the Chomsky hierarchy can be expressed in a unification-based grammar as we have defined them. Given that they are so powerful, unification-based grammars are still of interest to computational linguists since, subject to certain restrictions, they can still be processed efficiently.

Unification with Other Formalisms. Feature structures have been introduced into other formalisms, resulting in a large family of unification-based formalisms. For example, they have been introduced into CG, resulting in unification categorial grammar and categorial unification grammar (14). They have been introduced into TAGs, resulting in feature-structure-based TAG (16). Unification also plays an important role in logic grammars, which have been explored in logic programming (see LOGIC PROGRAMMING).

PROCESSING

The actual grammars developed within a particular grammar formalism can be used in the analysis and generation of natural language text. The task of assigning a structure to a given input text is known as parsing, while the process of determining the text associated with an underlying structure is known as generation. In principle, the same grammar could be used for both analysis and generation of text, but in practice there are different factors that come into play that make the use of a single grammar for these two tasks more difficult (27). The actual algorithms used may differ from formalism to formalism, but there are many underlying principles that are the same regardless of the formalism used. Parsing and generation can both incorporate some specific techniques, which can be discussed independent of any specific formalism.

Parsing

A traditional parser is given a string of words and determines how the grammar rules can be used to produce a structure (like a tree in the case of a CFG) showing the syntactic dependences among words and various more complex constituents. Some general overviews of parsing can be found in textbooks (8,9,28). Parsing algorithms can be distinguished according to whether they work predominantly in a bottom-up or a top-down manner.

Bottom-Up Parsing. A bottom-up parser builds a parse tree starting from the leaves and proceeding up to the root. For example, given the string of words *The cat likes the dog,* the parser could first determine which rules are associated with each of the individual words, such as common_noun and verb, then determine which rules can be used to combine the constituents (like noun_phrase and verb_phrase), and finally determine that these higher-level constituents are licensed by the sentence rule to produce the complete structure. As we have described the process here, the parsing occurs bottom-up in a breadth-first fashion: One layer of the tree is completed before the next layer is attempted. The context-free style grammar rules are thus used in reverse during the parsing process: When symbols corresponding to the right-hand side of a rule are found, a symbol from the left-hand side of the grammar rule can be introduced into the parse tree. A depth-first approach places priority on the vertical construction of the parse tree rather than on the horizontal priority associated with breadth-first. For example, a bottom-up depth-first parser could see the constituent for the noun_phrase being completed before processing of the words *likes the dog* even began. One can even imagine variations depending on the order that the sentence itself is traversed: left to right versus right to left.

Top-Down Parsing. A top-down parser works from the start symbol and builds the tree downward toward the leaves (words). Context-free style rules are processed in the forward (left to right) direction: Appearance in the parse tree of a symbol from the left-hand side of a grammar rule licenses the introduction of the symbols from the right-hand side, subject to the condition that they do not introduce terminal symbols that are inconsistent with the sentence being processed. Again, this can be done in a depth-first or breadth-first manner.

Dealing with Ambiguity. Ambiguity is perhaps the greatest stumbling block that parsers encounter. Natural languages, unlike artificial languages like computer programming languages, are inherently ambiguous. So, frequently a parser must choose concerning which rule is responsible for a constituent or which dictionary entry is appropriate for a word. If the correct choice is not made initially, then the parser must later go back and try the other alternatives. Clearly, as sentences get longer, the combinations of possible choices increase, leading to potentially huge numbers of interpretations for sentences.

Performance. The performance of parsers varies depending on the actual grammar and text being processed. For instance, when processing ungrammatical sentences, the parser must exhaust all possible rule combinations before it can conclude that the sentence is not grammatical. Some naive algorithms will not even terminate on certain grammar and sentence combinations. For CFGs, the most efficient algorithm takes an amount of time proportional to the cube of the sentence length in the worst case, so the task is of polynomial time complexity. There are polynomial time algorithms for TAGs as well (29). Some comparisons between different types of parsing algorithms can be found in Ref. 30. Worst-case complexity does not tell the complete story, since syntactic processing in natural language processing systems need not be done in isolation. For instance, semantic, pragmatic, and/or statistical information can be used in parallel to constrain the processing and obtain average-case performance appropriate for real-time systems.

The performance of various parsing algorithms will also vary depending on the kind of ambiguity found in grammars and lexicons, and on the structure of rules found in the grammar. In addition, there is also a trade-off of time and space, with more efficient processing resulting at a cost of increased memory usage.

Generation

In generation, the goal is to create a sentence from some underlying structure. There is a great variety of generation algorithms depending not only on the formalism used but on the kind of underlying structure used (31). The underlying structure typically incorporates some aspect of the meaning (semantics) of the sentence. While the parsing algorithms were required to deal with multiple alternatives in the form of ambiguity, generation algorithms must choose which sentence to generate from a selection of acceptable alternatives. This is not just simply the mirror image of the ambiguity encountered during parsing. Consider the problem of generating a sentence saying that Fido loves Bill. One could imagine generating any of the following: (1) *Fido loves Bill,* (2) *It is Bill that Fido loves,* (3) *Bill, Fido loves,* (4) *Bill is loved by Fido,* and (5) *Who Fido loves is Bill.* For a generator to choose between these alternatives requires a great deal of subtle information to be included in the underlying structure.

Memoization

Given the large number of choices involved in the parsing and generation processes, it is important for all but the shortest sentences to ensure that correct or even incorrect hypotheses related to the construction of certain subconstituents are "remembered" in case they are again encountered during alternative analyses. Previously encountered hypotheses or constituents are frequently stored in a chart or table. The storage of such results is often referred to as memoization since it can be viewed as the processor writing a "memo" concerning the result obtained (32). There is a cost in terms of space and time associated with the storage, and some algorithms thus do not incorporate such techniques. For example, a backtracking algorithm keeps track of all possible choices at a decision point, and when a parse or generation attempt fails, the process reverts back to the state at the most recent decision point and the next choice is tried. This frequently results in a duplication of work as the processor again moves forward from the decision point. By using a chart or table, some intermediate results obtained during a parse or generation attempt for a complex constituent that failed can be reused in an alternative attempt.

Compilation

Some parsers and generators work directly with grammars associated with a specific formalism, while others use grammars that have been compiled or translated into a different form. One reason for converting a grammar from one form into another is that it can result in more efficient or faster processing of natural language. Grammar designers can describe linguistic knowledge using a high-level grammar formalism, and the resulting grammar can then be compiled down into a low-level representation that can be processed easily by the computer, as is done in Ref. 33. Sometimes this compilation process may result in a representation that only approximates the original grammar but that may be sufficient for the task at hand. For example, one could imagine a CCG being compiled down into a CFG under the assumption that the resulting low-level CFG does not incorporate any grammar rules for categories containing more than n slashes (for some value of n). One could also imagine compiling a CFG down into a regular grammar under the assumption that the resulting grammar will not incorporate analysis for constructions involving more than m levels of recursion in the original grammar. There has even been work done on converting HPSG representations into TAG representations (34).

There are also disadvantages associated with compiling grammars into a lower-level representation rather than using them directly. First, the compilation process can be very intensive in time and space if the high-level and low-level representations differ considerably. Second, it can be difficult to diagnose problems associated with the processes on the low-level representations when they differ significantly from the high-level representations—the compilation process can cause substantial changes to the representations.

BIBLIOGRAPHY

1. W. J. Hutchins and H. L. Somers, *An Introduction to Machine-Translation,* San Diego: Academic Press, 1992.
2. I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, Natural language interfaces to databases—an introduction, *J. Nat. Language Eng.,* **1**: 29–81, 1994.
3. L. R. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition,* Englewood Cliffs, NJ: Prentice-Hall, 1993.
4. E. Charniak, *Statistical Language Learning,* Cambridge, MA: MIT Press, 1993.
5. S. Young and G. Bloothooft (eds.), *Corpus-Based Methods in Language and Speech Processing,* Boston: Kluwer, 1997.
6. B. H. Partee, A. ter Meulen, and R. E. Wall, *Mathematical Methods in Linguistics,* Boston: Kluwer, 1993.
7. W. J. Savitch et al. (eds.), *The Formal Complexity of Natural Language,* Boston: D. Reidel, 1987.
8. R. Grishman, *Computational Linguistics,* New York: Cambridge Univ. Press, 1986.
9. M. A. Covington, *Natural Language Processing for Prolog Programmers,* Englewood Cliffs, NJ: Prentice-Hall, 1994.
10. G. D. Ritchie et al., *Computational Morphology,* Cambridge, MA: MIT Press, 1992.
11. D. E. Johnson and L. S. Moss, Mathematics of Language, Special Issue of *Ling. Philos.* **20**: 571–756, 1997.
12. N. Sager, *Natural Language Information Processing,* Reading, MA: Addison-Wesley, 1981.
13. N. Chomsky, *The Logical Structure of Linguistic Theory,* New York: Plenum, 1975.
14. M. M. Wood, *Categorical Grammars,* New York: Hudson, 1993.
15. M. Steedman, *Surface Structure and Interpretation,* Cambridge, MA: MIT Press, 1996.
16. A. K. Joshi, K. Vijay-Shanker, and D. Weir, The convergence of mildly context-sensitive grammar formalisms, in P. Sells, S. M. Shieber, and T. Wasow (eds.), *Foundational Issues in Natural Language Processing,* Cambridge, MA: MIT Press, 1991.
17. C. J. Pollard, *Generalized Phrase Structure Grammars, Head Grammars and Natural Language,* Ph.D. dissertation, Department of Linguistics, Stanford University, Stanford, CA, 1984.
18. C. J. Pollard and I. Sag, *Head-Driven Phrase Structure Grammar,* Chicago: University of Chicago Press, 1994.
19. K. Roach, Formal properties of head grammars, in A. Manaster-Ramer (ed.), *Mathematics of Language,* Amsterdam: John Benjamins, 1987.

20. A. K. Joshi, An introduction to tree adjoining grammars, in A. Manaster-Ramer (ed.), *The Mathematics of Language,* Amsterdam: John Benjamins, 1987.
21. A. Radford, *Transformational Grammar: A First Course,* New York: Cambridge Univ. Press, 1988.
22. G. Gazdar, Applicability of index grammars to natural languages, in U.Reyle and C. Rohrer (eds.), *Natural Language Parsing and Linguistic Theories,* Boston: D. Reidel, 1988.
23. A. V. Aho, Indexed grammars: An extension of the context-free grammars, *J. ACM,* **15**: 641–671, 1968.
24. G. Gazdar et al., *Generalized Phrase Structure Grammar,* Oxford: Basil Blackwell, 1985.
25. U. Reyle and C. Rohrer (eds.), *Natural Language Parsing and Linguistic Theories,* Boston: D. Reidel, 1988.
26. B. Carpenter, *The Logic of Typed Feature Structures,* New York: Cambridge Univ. Press, 1992.
27. T. Strzalkowski (ed.), *Reversible Grammar in Natural Language Processing,* Boston: Kluwer, 1993.
28. F. C. N. Pereira and S. M. Shieber, *Prolog and Natural Language Analysis,* CSLI Lecture Notes, Chicago: Univ. Chicago Press, 1987.
29. S. Rajasekaran, Tree-Adjoining Language Parsing in $o(n^6)$ Time, *SIAM J. Comput.,* **25**: 862–873, 1996.
30. G. van Noord, An efficient implementation of the head-corner parser, *Computational Linguistics,* **23**: 425–456, 1997.
31. D. D. McDonald and L. Bolc (eds.), *Natural Language Generation Systems,* New York: Springer-Verlag, 1988.
32. M. Johnson, Memoization in top-down parsing, *Computational Lingustics,* **21**: 405–417, 1995.
33. R. C. Moore et al., CommandTalk: A spoken-language interface for battlefield simulations, *Proc. 5th Conf. Appl. Nat. Language Proc.,* Association for Computational Linguistics, San Francisco: Morgan Kaufmann Publishers, 1997, pp. 1–7.
34. R. Kasper et al., Compilation of HPSG to TAG, *Proc. 33rd Annu. Meet. of the Assoc. for Computational Linguistics,* San Francisco: Morgan Kaufmann Publishers, 1995, pp. 92–99.

FRED POPOWICH
Simon Fraser University

COMPUTATIONAL NUMBER THEORY. See NUMBER THEORY.

COMPUTED TOMOGRAPHY. See COMPUTERIZED TOMOGRAPHY; TOMOGRAPHY.

COMPUTER-AIDED DESIGN. See CAD FOR MANUFACTURABILITY OF INTEGRATED CIRCUITS.

COMPUTER-AIDED DESIGN FOR FPGA. See CAD FOR FIELD PROGRAMMABLE GATE ARRAYS.