MODULE 6

****

SYLLABUS

Module 6 (9Hrs 20%Marks)

Expert Systems: rule based expert systems. Natural language processing-natural language understanding problem, deconstructing language. Syntax stochastic tools for language analysis, natural language applications

****

Expert Systems

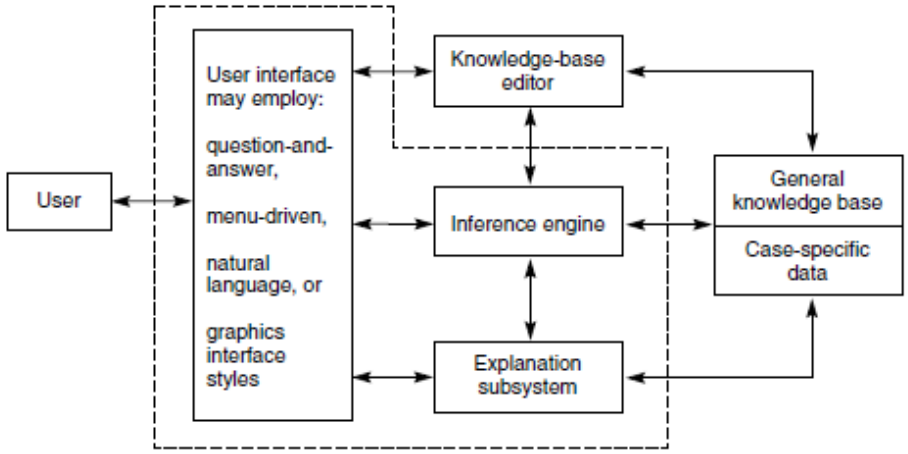

Figure 8.1 Architecture of a typical expert system for a particular problem domain.

The user interacts with the system through a *user interface* that simplifies communication and hides much of the complexity, such as the internal structure of the rule base. Expert system interfaces employ a variety of user styles, including question-and-answer, menu-driven, or graphical interfaces. The final decision on the interface type is a compromise between user needs and the requirements of the knowledge base and inferencing system.

The heart of the expert system is the *knowledge base*, which contains the knowledge of a particular application domain. In a rule-based expert system this knowledge is most often represented in the form of *if... then...* rules. The knowledge base contains both general knowledge as well as case-specific information.

The inference engine applies the knowledge to the solution of actual problems. It is essentially an interpreter for the knowledge base. In the production system, the inference engine performs the recognize-act control cycle. The procedures that implement the control cycle are separate from the production rules themselves. It is important to maintain this separation of the knowledge base and inference engine for several reasons:

1. This separation makes it possible to represent knowledge in a more natural fashion. If ... then... rules, for example, are closer to the way in which humans describe their problem-solving skills than is lower-level computer code.
2. Because the knowledge base is separated from the program's lower-level control structures, expert system builders can focus on capturing and organizing problem- solving knowledge rather than on the details of its computer implementation.
3. Ideally, the separation of knowledge and control allows changes to be made in one part of the knowledge base without creating side effects in others.
4. The separation of the knowledge and control elements of the program allows the same control and interface software to be used in a variety of systems. The expert system shell has all the components of Figure above except that the knowledge base and case-specific data are empty and can be added for a new application. The broken lines of Figure indicate the shell modules.

The expert system must keep track of case-specific data: the facts, conclusions, and other information relevant to the case under consideration. This includes the data given in a problem instance, partial conclusions, confidence measures of conclusions, and dead ends in the search process. This information is separate from the general knowledge base.

The explanation subsystem allows the program to explain its reasoning to the user. These explanations include justifications for the system's conclusions, in response to how queries, explanations of why the system needs a particular piece of data, why queries, and, where useful, tutorial explanations or deeper theoretical justifications of the program's actions.

Knowledge-base editors help the programmer locate and correct bugs in the program's performance, often accessing the information provided by the explanation subsystem. They also may assist in the addition of new knowledge, help maintain correct rule syntax, and perform consistency checks on any updated knowledge base.

Selecting a Problem and the Knowledge Engineering Process

Expert systems involve a considerable investment of money and human effort. Attempts to solve a problem that is too complex, too poorly understood, or otherwise unsuited to the available technology can lead to costly and embarrassing failures. Researchers have developed guidelines to determine whether a problem is appropriate for expert system solution:

1. **The need for the solution justifies the cost and effort of building an expert system.** Many expert systems have been built in domains such as mineral

exploration, business, defense, and medicine where a large potential exists for savings in terms of money, time, and human life.

2. **Human expertise is not available in all situations where it is needed**. In geology, for example, there is a need for expertise at remote mining and drilling sites. Often, geologists and other engineers find themselves traveling large distances to visit sites, with resulting expense and wasted time. By placing expert systems at remote sites, many problems may be solved without needing a visit.
3. **The problem may be solved using symbolic reasoning**. Problem solutions should not require physical dexterity or perceptual skill. Robots and vision systems currently lack the sophistication and flexibility of humans.
4. **The problem domain is well structured and does not require common sense reasoning**. Highly technical fields have the advantage of being well studied and formalized: terms are well defined and domains have clear and specific conceptual models. In contrast, common sense reasoning is difficult to automate.
5. **The problem may not be solved using traditional computing methods.** Expert system technology should not be used where unnecessary. If a problem can be solved satisfactorily using more traditional techniques, then it is not a candidate.
6. **Cooperative and articulate experts exist.** The knowledge used by expert systems comes from the experience and judgment of humans working in the domain. It is important that these experts be both willing and able to share knowledge.
7. **The problem is of proper size and scope.** For example, a program that attempted to capture all of the expertise of a medical doctor would not be feasible; a program that advised MDs on the use of a particular piece of diagnostic equipment or a particular set of diagnoses would be more appropriate.

The primary people involved in building an expert system are the knowledge engineer, the domain expert, and the end user. The knowledge engineer is the AI language and representation expert. His or her main task is to select the software and hardware tools for the project, help the domain expert articulate the necessary knowledge, and implement that knowledge in a correct and efficient knowledge base. Often, the knowledge engineer is initially ignorant of the application domain.

The domain expert provides the knowledge of the problem area. The domain expert is generally someone who has worked in the domain area and understands its problem-solving techniques, such as shortcuts, handling imprecise data, evaluating partial solutions, and all the other skills that mark a person as an expert problem solver. The domain expert is primarily responsible for spelling out these skills to the knowledge engineer.

As in most applications, the end user determines the major design constraints. Unless the user is happy, the development effort is by and large wasted. Like most AI programming, building expert systems requires a nontraditional development cycle

based on early prototyping and incremental revision of the code. Generally, work on the system begins with the knowledge engineer attempting to gain some familiarity with the problem domain. This helps in communicating with the domain expert. This is done in initial interviews with the expert and by observing experts during the performance of their job. Next, the knowledge engineer and expert begin the process of extracting the expert's problem-solving knowledge. This is often done by giving the domain expert a series of sample problems and having him or her explain the techniques used in their solution. Video and/or audio tapes are often essential for capturing this process.

It is often useful for the knowledge engineer to be a novice in the problem domain. Human experts are notoriously unreliable in explaining exactly what goes on in solving a complex problem. Often they forget to mention steps that have become obvious or even automatic to them after years of work in their field. Knowledge engineers, by virtue of their relative naivete in the domain, can spot these conceptual jumps and ask for help.

Once the knowledge engineer has obtained a general overview of the problem domain and gone through several problem-solving sessions with the expert, he or she is ready to begin actual design of the system: selecting a way to represent the knowledge, such as rules or frames, determining the search strategy, forward, backward, depth-first, best-first etc., and designing the user interface. After making these design commitments, the knowledge engineer builds a prototype.

This prototype should be able to solve problems in a small area of the domain and provide a test bed for preliminary design assumptions. Once the prototype has been implemented, the knowledge engineer and domain expert test and refine its knowledge by giving it problems to solve and correcting its shortcomings. Should the assumptions made in designing the prototype prove correct, the prototype can be incrementally extended until it becomes a final system.

Expert systems are built by progressive approximations, with the program's mistakes leading to corrections or additions to the knowledge base. In a sense, the knowledge base is "grown" rather than constructed. Figure below presents a flow chart describing the exploratory programming development cycle.

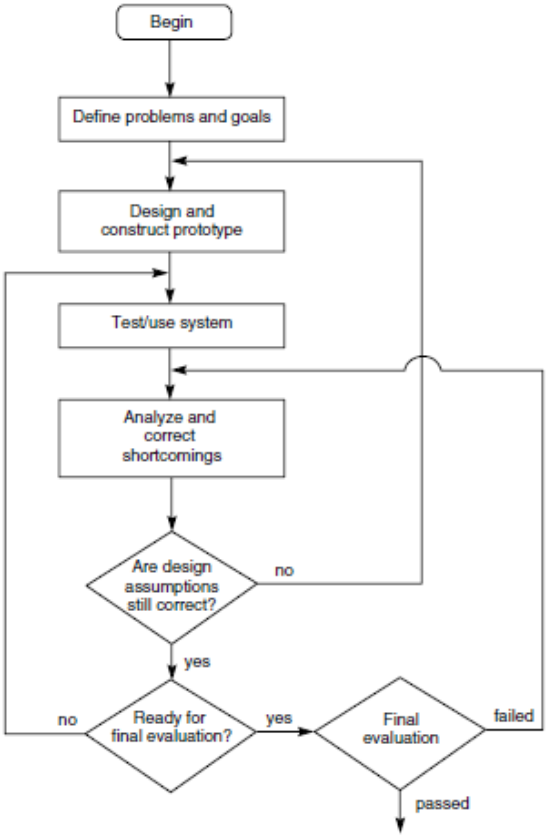

Figure 8.2 Exploratory development cycle.

The second major feature of expert system programming is that the program need never be considered "finished." A large heuristic knowledge base will always have limitations. The modularity of the production system model make it natural to add new rules or make up for the shortcomings of the present rule base at any time.

Conceptual Models and Their Role in Knowledge Acquisition

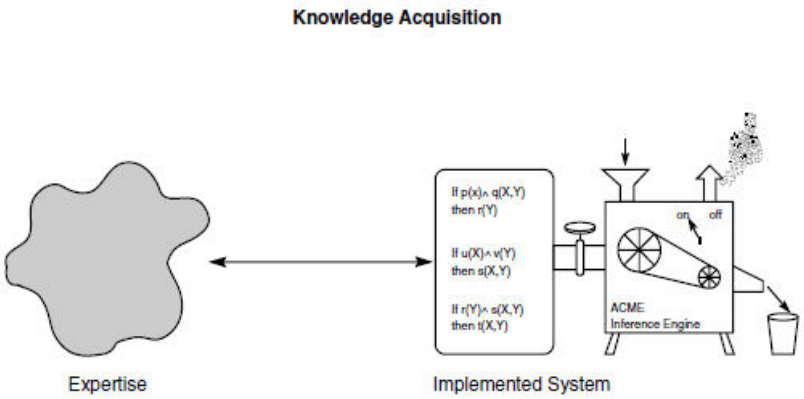

Figure 8.3 The standard view of building an expert system.

The human expert, working in an application area, operates in a domain of knowledge, skill, and practice. This knowledge is often vague, imprecise, and only partially verbalized. The knowledge engineer must translate this informal expertise into a formal language suited to a computational system. A number of important issues arises in the process of formalizing human skilled performance:

1. Human skill is often inaccessible to the conscious mind. As Aristotle points out in his Ethics, "what we have to learn to do, we learn by doing." Skills such as those possessed by medical doctors are learned as much in years of internship and residency, with their constant focus on patients, as they are in physiology lectures, where emphasis is on experiment and theory. Delivery of medical care is to a great extent practice-driven. After years of performance these skills become highly integrated and function at a largely unconscious level. It may be difficult for experts to describe exactly what they are doing in problem solving.
2. Human expertise often takes the form of knowing how to cope in a situation rather than knowing what a rational characterization of the situation might be, of developing skilled performance mechanisms rather than a fundamental understanding of what these mechanisms are. An obvious example of this is riding a unicycle: the successful rider is not, in real time, consciously solving multiple sets of simultaneous differential equations to preserve balance; rather she is using an intuitive combination of feelings of "gravity," "momentum," and "inertia" to form a usable control procedure.
3. We often think of knowledge acquisition as gaining factual knowledge of an objective reality, the so-called "real world". As both theory and practice have shown, human expertise represents an individual's or a community's model of the world. Such models are as influenced by convention, social processes, and hidden agendas as they are by empirical methodologies.
4. Expertise changes. Not only do human experts gain new knowledge, but also existing knowledge may be subject to radical reformulation, as evidenced by ongoing controversies in both scientific and social fields.

Consequently, knowledge engineering is difficult and should be viewed as spanning the life cycle of any expert system. To simplify this task, it is useful to consider, as in Figure below, a conceptual model that lies between human expertise and the implemented program.

By a conceptual model, we mean the knowledge engineer's evolving conception of the domain knowledge. Although this is undoubtedly different from the domain expert's, it is this model that actually determines the construction of the formal knowledge base.

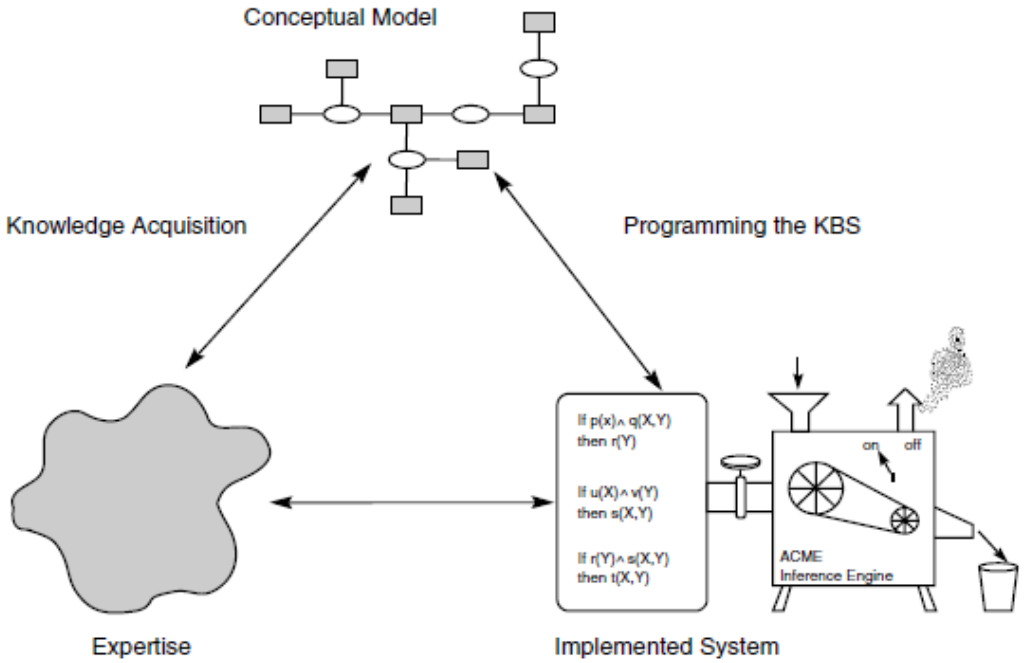

Figure 8.4 The role of mental or conceptual models in problem solving.

Because of the complexity of most interesting problems, we should not take this intermediate stage for granted. Knowledge engineers should document and make public their assumptions about the domain through common software engineering methodologies. An expert system should include a requirements document; however, because of the constraints of exploratory programming, expert system requirements should be treated as coevolving with the prototype. Data dictionaries, graphic representations of state spaces, and comments in the code itself are all part of this model. By publicizing these design decisions, we reduce errors in both the implementation and maintenance of the program.

The conceptual model is not formal or directly executable on a computer. It is an intermediate design construct, a template to begin to constrain and codify human skill. It can, if the knowledge engineer uses a predicate calculus model, begin as a number of simple networks representing states of reasoning through typical problem-solving situations. Only after further refinement does this network become explicit if... then... rules.

Rule-Based Expert Systems

Rule-based expert systems represent problem-solving knowledge as if... then... rules. This approach is one of the oldest techniques for representing domain knowledge in an expert system. It is also one of the most natural, and remains widely used in practical and experimental expert systems.

In a rule-based system, condition action pairs are represented as if... then... rules, with the premises of the rules, the if portion, corresponding to the condition, and the

conclusion, the then portion, corresponding to the action: when the condition is satisfied, the expert system takes the action of asserting the conclusion as true. Case-specific data can be kept in the working memory. The inference engine implements the recognize-act cycle of the production system; this control may be either data-driven or goal-driven.

In a goal-driven expert system, the goal expression is initially placed in working memory. The system matches rule conclusions with the goal, selecting one rule and placing its premises in the working memory. This corresponds to a decomposition of the problem's goal into simpler subgoals. The process continues in the next iteration of the production system, with these premises becoming the new goals to match against rule conclusions. The system thus works back from the original goal until all the subgoals in working memory are known to be true, indicating that the hypothesis has been verified. Thus, backward search in an expert system corresponds roughly to the process of hypothesis testing in human problem solving.

In an expert system, subgoals can be solved by asking the user for information. Some expert systems allow the system designer to specify which subgoals may be solved by asking the user. Others simply ask the user about any subgoals that fail to match rules in the knowledge base; i.e., if the program cannot infer the truth of a subgoal, it asks the user.

As an example of goal-driven problem solving with user queries, we next offer a small expert system for analysis of automotive problems. This is not a full diagnostic system, as it contains only four very simple rules. It is intended as an example to demonstrate goal-driven rule chaining, the integration of new data, and the use of explanation facilities:

Rule 1: if

the engine is getting gas, and

the engine will turn over,

then

the problem is spark plugs.

Rule 2: if

the engine does not turn over, and

the lights do not come on

then

the problem is battery or cables.

Rule 3: if

the engine does not turn over, and

the lights do come on

then

the problem is the starter motor.

Rule 4: if

there is gas in the fuel tank, and

there is gas in the carburetor

then

the engine is getting gas.

To run this knowledge base under a goal-directed control regime, place the top-level goal, the problem is X, in working memory as shown in Figure below. X is a variable that can match with any phrase, as an example, the problem is battery or cables; it will become bound to the solution when the problem is solved.

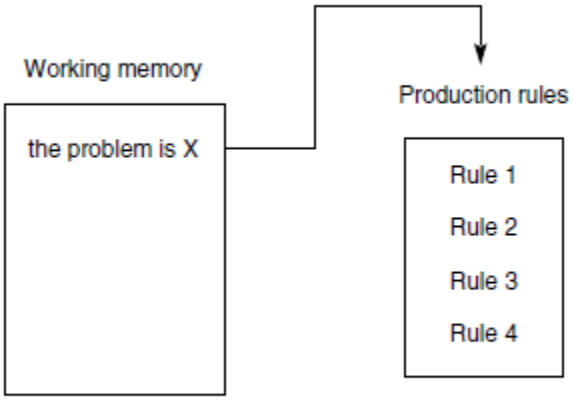

Figure 8.5 The production system at the start of a consultation in the car diagnostic example.

Three rules match with this expression in working memory: rule 1, rule 2, and rule 3. If we resolve conflicts in favor of the lowest-numbered rule, then rule 1 will fire. This causes X to be bound to the value spark plugs and the premises of rule 1 to be placed in the working memory as in Figure below. The system has thus chosen to explore the possible hypothesis that the spark plugs are bad. Another way to look at this is that the system has selected an or branch in an and/or graph.

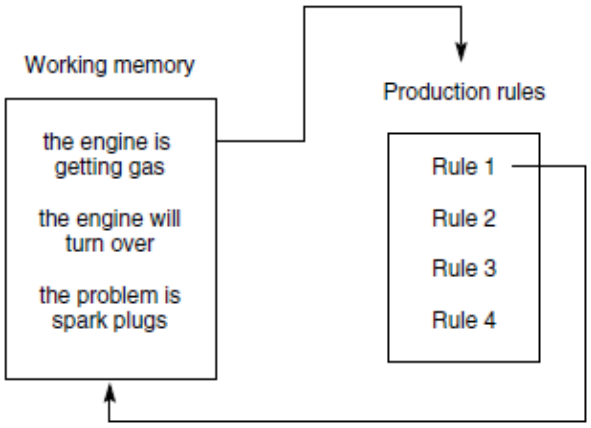

Figure 8.6 The production system after Rule 1 has fired.

Note that there are two premises to rule 1, both of which must be satisfied to prove the conclusion true. These are and branches of the search graph representing a decomposition of the problem (finding whether the problem is spark plugs) into two subproblems (finding whether the engine is getting gas and whether the engine will turn over). We may then fire rule 4, whose conclusion matches with the engine is getting gas, causing its premises to be placed in working memory as in Figure below

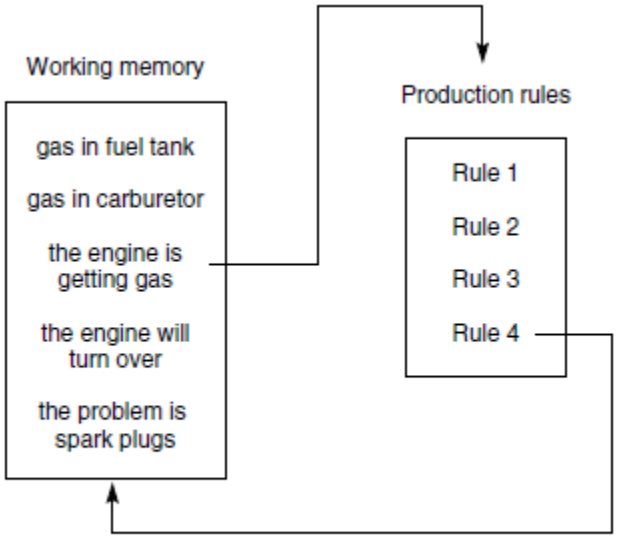

Figure 8.7 The system after Rule 4 has fired. Note the stack-based approach to goal reduction.

At this point, there are three entries in working memory that do not match with any rule conclusions. Our expert system will, in this situation, query the user directly about these subgoals. If the user confirms all three of these as true, the expert system will have successfully determined that the car will not start because the spark plugs are bad. In finding this solution, the system has searched the leftmost branch of the and/or graph presented in Figure below

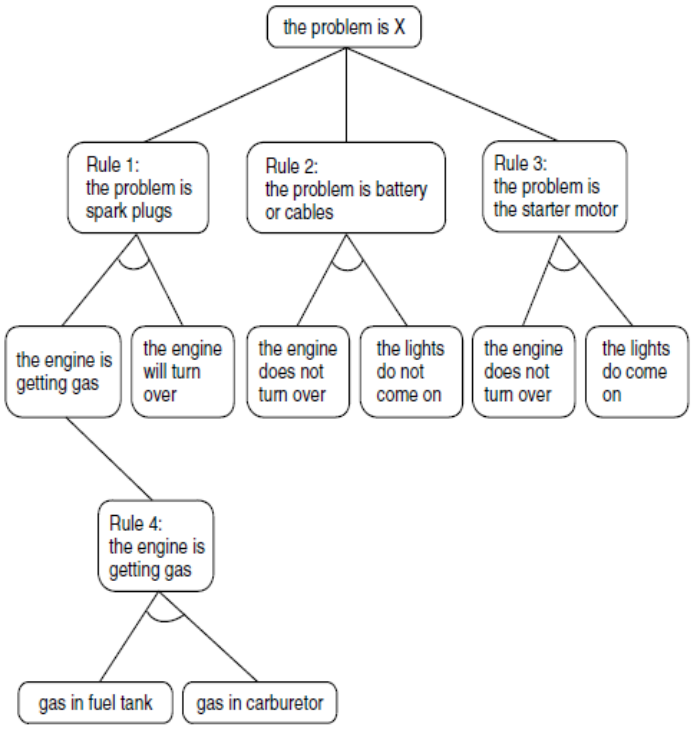

Figure 8.8 The and/or graph searched in the car diagnosis example, with the conclusion of Rule 4 matching the first premise of Rule 1.

This is, of course, a very simple example. Not only is its automotive knowledge limited at best, but it also ignores a number of important aspects of real implementations. The rules are phrased in English, rather than a formal language. On finding a solution, a real expert system will tell the user its diagnosis (our model simply stops). Also, we should maintain enough of a trace of the reasoning to allow backtracking if necessary. In our example, had we failed to determine that the spark plugs were bad, we would have needed to back up to the top level and try rule 2 instead.

The Natural Language Understanding Problem

Communicating with natural language, whether as text or as speech, depends heavily on our language skills, knowledge of the domain of interest, and expectations within that domain. Understanding language is not merely the transmission of words: it also requires inferences about the speaker's goals, knowledge, and assumptions, as well as about the context of the interaction. Implementing a natural language understanding program requires that we represent knowledge and expectations of the domain and reason effectively about them. We must consider such issues as nonmonotonicity, belief revision, metaphor, planning, learning, and the practical complexities of human interaction. But these are the central problems of artificial intelligence itself!

We cannot merely chain together the dictionary meanings of Shakespeare's words and call the result understanding. Instead, we must employ a complex process of

capturing the word patterns, parsing the sentences, constructing a representation of the semantic meanings, and interpreting these meanings in light of our knowledge of the problem domain.

There are (at least!) three major issues involved in understanding language. First, a large amount of human knowledge is assumed. Language acts describe relationships in an often complex world. Knowledge of these relationships must be part of any understanding system. Second, language is pattern based: phonemes are components of words and words make phrases and sentences. Phoneme, word, and sentence orders are not random. Communication is impossible without a rather constrained use of these components. Finally, language acts are the product of agents, either human or computer. Agents are embedded in complex environments with both individual and sociological dimensions. Language acts are purposive.

Deconstructing Language: An Analysis

Language is a complicated phenomenon, involving processes as varied as the recognition of sounds or printed letters, syntactic parsing, high-level semantic inferences, and even the communication of emotional content through rhythm and inflection. To manage this complexity, linguists have described different levels of analysis for natural language:

1. **Prosody** deals with the rhythm and intonation of language. This level of analysis is difficult to formalize and often neglected; however, its importance is evident in the powerful effect of poetry or religious chants, as well as the role played by rhythm in children's wordplay and the babbling of infants.
2. **Phonology** examines the sounds that are combined to form language. This branch of linguistics is important for computerized speech recognition and generation.
3. **Morphology** is concerned with the components (morphemes) that make up words. These include the rules governing the formation of words, such as the effect of prefixes (un-, non-, anti-, etc.) and suffixes (-ing, -ly, etc.) that modify the meaning of root words. Morphological analysis is important in determining the role of a word in a sentence, including its tense, number, and part of speech.
4. **Syntax** studies the rules for combining words into legal phrases and sentences, and the use of those rules to parse and generate sentences. This is the best formalized and thus the most successfully automated component of linguistic analysis.
5. **Semantics** considers the meaning of words, phrases, and sentences and the ways in which meaning is conveyed in natural language expressions.
6. **Pragmatics** is the study of the ways in which language is used and its effects on the listener. For example, pragmatics would address the reason why "Yes" is usually an inappropriate answer to the question "Do you have a watch?"

7. **World knowledge** includes knowledge of the physical world, the world of human social interaction, and the role of goals and intentions in communication. This general background knowledge is essential to understand the full meaning of a text or conversation.

Although these levels of analysis seem natural and are supported by psychological evidence, they are, to some extent, artificial divisions that have been imposed on language. All of these interact extensively, with even low-level intonations and rhythmic variations having an effect on the meaning of an utterance, for example, the use of sarcasm. This interaction is evident in the relationship between syntax and semantics, and although some division along these lines seems essential, the exact boundary is difficult to characterize.

For example, sentences such as "They are eating apples" have multiple parsings, resolved only by attention to meaning in context. Syntax also affects semantics, as is seen by the role of phrase structure in interpreting the meaning of a sentence. Although the exact nature of the distinction between syntax and semantics is often debated, both the psychological evidence and its utility in managing the complexity of the problem argue for its retention.

Although the specific organization of natural language understanding programs varies with different philosophies and applications—e.g., a front end for a database, an automatic translation system, a story understanding program—all of them must translate the original sentence into an internal representation of its meaning. Generally, symbol-based natural language understanding follows the stages presented in Figure below

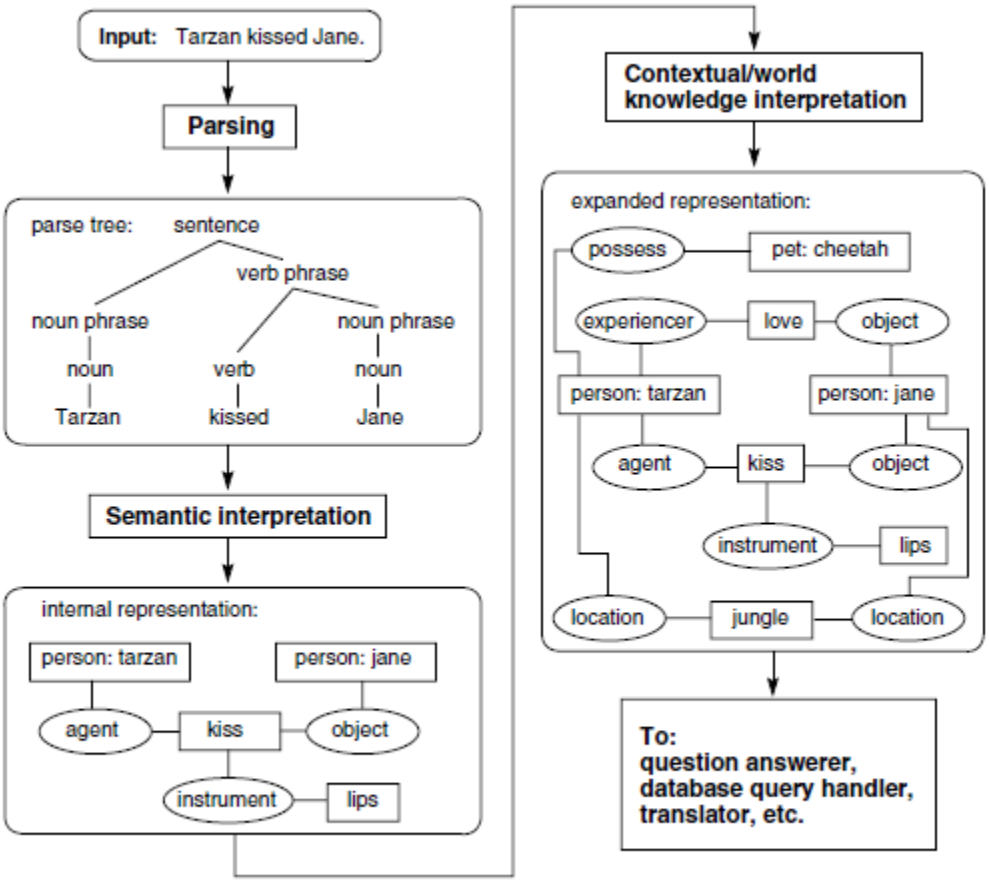

Figure 15.2 Stages in producing an internal representation of a sentence.

The first stage is parsing, which analyzes the syntactic structure of sentences. Parsing both verifies that sentences are syntactically well formed and also determines a linguistic structure. By identifying the major linguistic relations such as subject-verb, verb-object, and noun-modifier, the parser provides a framework for semantic interpretation. This is often represented as a parse tree. The language parser employs knowledge of language syntax, morphology, and some semantics.

The second stage is semantic interpretation, which produces a representation of the meaning of the text. Semantic interpretation uses knowledge about the meaning of words and linguistic structure, such as case roles of nouns or the transitivity of verbs. In the Figure the program used knowledge of the meaning of kiss to add the default value of lips for the instrument of kissing. This stage also performs semantic consistency checks. For example, the definition of the verb kiss may include constraints that the object be a person if the agent is a person, that is, Tarzan kisses Jane and does not (normally) kiss Cheetah.

In the third stage, structures from the knowledge base are added to the internal representation of the sentence to produce an expanded representation of the sentence's meaning. This adds the necessary world knowledge required for complete understanding, such as the facts that Tarzan loves Jane, that Jane and Tarzan

live in the jungle, and that Cheetah is Tarzan's pet. This resulting structure represents the meaning of the natural language text and is used by the system for further processing.

These stages exist in most (non probabilistic) natural language understanding systems, although they may or may not correspond to distinct software modules. For example, many programs do not produce an explicit parse tree but generate an internal semantic representation directly. Nevertheless, the tree is implicit in the parse of the sentence. Incremental parsing is a commonly used technique in which a fragment of the internal representation is produced as soon as a significant part of the sentence is parsed. These fragments are combined into a complete structure as the parse proceeds. These fragments are also used to resolve ambiguities and guide the parser.

Syntax

Specification and Parsing Using Context-Free Grammars

The rules listed below define a grammar for simple transitive sentences such as "The man likes the dog." The rules are numbered for reference.

1. sentence ↔ noun_phrase verb_phrase
2. noun_phrase ↔ noun
3. noun_phrase ↔ article noun
4. verb_phrase ↔ verb
5. verb_phrase ↔ verb noun_phrase
6. article ↔ a
7. article ↔ the
8. noun ↔ man
9. noun ↔ dog
10. verb ↔ likes
11. verb ↔ bites

Rules 6 through 11 have English words on the right-hand side; these rules form a dictionary of words that may appear in sentences. These words are the terminals of the grammar and define a lexicon of the language. Terms that describe higher-level linguistic concepts (sentence, noun_phrase, etc.) are called nonterminals. Nonterminals appear in this typeface. Note that terminals do not appear in the left-hand side of any rule.

A legal sentence is any string of terminals that can be derived using these rules. A derivation begins with the nonterminal symbol sentence and produces a string of terminals through a series of substitutions defined by the rules of the grammar. A legal substitution replaces a symbol that matches the left-hand side of a rule with the symbols on the righthand side of that rule. At intermediate stages of the derivation, the string may contain both terminals and nonterminals and is called a sentential form.

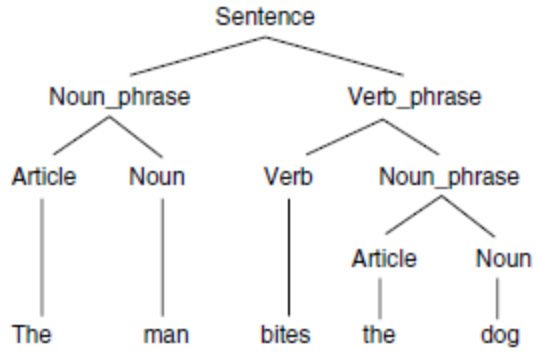

Figure 15.3 Parse tree for the sentence The man bites the dog.

A derivation of the sentence "The man bites the dog" is given by:

STRING	APPLY RULE #
sentence	1
noun_phrase verb_phrase	3
article noun verb_phrase	7
The noun verb_phrase	8
The man verb_phrase	5
The man verb noun_phrase	11
The man bites noun_phrase	3
The man bites article noun	7
The man bites the noun	9
The man bites the dog	

This is an example of a top-down derivation: it begins with the sentence symbol and works down to a string of terminals. A bottom-up derivation starts with a string of terminals and replaces right-hand-side patterns with those from the left-hand side, terminating when all that remains is the sentence symbol.

A derivation can be represented as a tree structure, known as a parse tree, in which each node is a symbol from the set of rules of the grammar. The tree's interior nodes are nonterminals; each node and its children correspond, respectively, to the left- and righthand side of a rule in the grammar. The leaf nodes are terminals and the sentence symbol is the root of the tree.

Not only does the existence of a derivation or parse tree prove that a sentence is legal in the grammar, but it also determines the structure of the sentence. The phrase structure of the grammar defines the deeper linguistic organization of the language. For example, the breakdown of a sentence into a noun_phrase and a verb_phrase specifies the relation between an action and its agent. This phrase structure plays an essential role in semantic interpretation by defining intermediate stages in a derivation at which semantic processing may take place.

Parsing is the problem of constructing a derivation or a parse tree for an input string from a formal definition of a grammar. Parsing algorithms fall into two classes: top-down parsers, which begin with the top-level sentence symbol and attempt to build a tree whose leaves match the target sentence, and bottom-up parsers, which start with the words in the sentence (the terminals) and attempt to find a series of reductions that yield the sentence symbol.

One difficulty that can add huge complexity to the parsing problem, is in determining which of several potentially applicable rules should be used at any step of the derivation. If the wrong choice is made, the parser may fail to recognize a legal sentence. For example, in attempting to parse the sentence "The dog bites" in a bottom-up fashion, rules 7, 9, and 11 produce the string article noun verb. At this point, an erroneous application of rule 2 would produce article noun_phrase verb; this could not be reduced to the sentence symbol. The parser should have used rule 3 instead. Similar problems can occur in a top-down parse.

The inverse problem is generation, or producing legal sentences from an internal representation. Generation starts with a representation of some meaningful content (such as a semantic network or conceptual dependency graph) and constructs a grammatically correct sentence that communicates this meaning. Generation is not merely the reverse of parsing; it encounters unique difficulties and requires separate methodologies. We present recursive descent context free and context sensitive parsers in the auxiliary materials.

Because parsing is particularly important in the processing of programming languages as well as natural language, researchers have developed a number of different parsing algorithms, including both top-down and bottom-up strategies. Although a complete survey of parsing algorithms is beyond the scope of this chapter, we do consider several approaches to parsing in some detail. In the next section we show the Earley parser, an important polynomial-time parser based on dynamic programming.