PART 4
Perception, Communication, and Expert Systems

12 Natural Language Processing

Perception and communication are essential components of intelligent behavior. They provide the ability to effectively interact with our environment. Humans perceive and communicate through their five basic senses of sight, hearing, touch, smell, and taste, and their ability to generate meaningful utterances. Two of the senses, sight and hearing are especially complex and require concious inferencing. Developing programs that understand natural language and that comprehend visual scenes are two of the most difficult tasks facing AI researchers.

Developing programs that understand a natural language is a difficult problem. Natural languages are large. They contain an infinity of different sentences. No matter how many sentences a person has heard or seen, new ones can always be produced. Also, there is much ambiguity in a natural language. Many words have several meanings such as can, bear, fly, and orange, and sentences can have different meanings in different contexts. This makes the creation of programs that "understand" a natural language, one of the most challenging tasks in AI. It requires that a program transform sentences occurring as part of a dialog into data structures which convey the intended meaning of the sentences to a reasoning program. In general, this means that the reasoning program must know a lot about the structure of the language, the possible semantics, the beliefs and goals of the user, and a great deal of general world knowledge.

12.1 INTRODUCTION

Developing programs to understand natural language is important in AI because a natural form of communication with systems is essential for user acceptance. Furthermore, one of the most critical tests for intelligent behavior is the ability to communicate effectively. Indeed, this was the purpose of the test proposed by Alan Turing (see Chapter 2). AI programs must be able to communicate with their human counterparts in a natural way, and natural language is one of the most important mediums for that purpose.

Before proceeding further, a definition of understanding as used here should be given. We say a program *understands* a natural language if it behaves by taking a (predictably) correct or acceptable action in response to the input. For example, we say a child demonstrates understanding if it responds with the correct answer to a question. The action taken need not be an external response. It may simply be the creation of some internal data structures as would occur in learning some new facts. But in any case, the structures created should be meaningful and correctly interact with the world model representation held by the program. In this chapter we explore many of the important issues related to natural language understanding and language generation.

12.2 OVERVIEW OF LINGUISTICS

An understanding of linguistics is not a prerequisite to the study of natural language understanding, but a familiarity with the basics of grammar is certainly important. We must understand how words and sentences are combined to produce meaningful word strings before we can expect to design successful language understanding systems. In a natural language, the sentence is the basic language element. A sentence is made up of words which express a complete thought. To express a complete thought, a sentence must have a subject and a predicate. The subject is what the sentence is about, and the predicate says something about the subject.

Sentences are classified by structure and usage. A simple sentence has one independent clause comprised of a subject and predicate. A compound sentence consists of two or more independent clauses connected by a conjunction or a semicolon. A complex sentence consists of an independent clause and one or more dependent clauses. Sentences are used to assert, query, and describe. The way a sentence is used determines its mood, declarative, imperative, interrogative, or exclamatory.

A word functions in a sentence as a part of speech. Parts of speech for the English language are nouns, pronouns, verbs, adjectives, adverbs, prepositions, conjunctions, and interjections.

A noun is a name for something (person, place, or thing). Pronouns replace nouns when the noun is already known. Verbs express action, being, or state of being. Adjectives are used to modify nouns and pronouns, and adverbs modify verbs, adjectives, or other adverbs. Prepositions establish the relationship between a noun and some other part of the sentence. Conjunctions join words or groups of words together, and interjections are used to express strong feelings apart from the rest of the sentence.

Phrases are made up of words but act as a single unit within a sentence. These form the building blocks for the syntactic structures we consider later.

Levels of Knowledge Used in Language Understanding

A language understanding program must have considerable knowledge about the structure of the language including what the words are and how they combine into phrases and sentences. It must also know the meanings of the words and how they contribute to the meanings of a sentence and to the context within which they are being used. Finally, a program must have some general world knowledge as well as knowledge of what humans know and how they reason. To carry on a conversation with someone requires that a person (or program) know about the world in general, know what other people know, and know the facts pertaining to a particular conversational setting. This all presumes a familiarity with the language structure and a minimal vocabulary.

The component forms of knowledge needed for an understanding of natural language are sometimes classified according to the following levels.

Phonological. This is knowledge which relates sounds to the words we recognize. A phoneme is the smallest unit of sound. Phones are aggregated into word sounds.

Morphological. This is lexical knowledge which relates to word constructions from basic units called morphemes. A morpheme is the smallest unit of meaning: for example, the construction of friendly from the root *friend* and the suffix *ly*.

Syntactic. This knowledge relates to how words are put together or structured to form grammatically correct sentences in the language.

Semantic. This knowledge is concerned with the meanings of words and phrases and how they combine to form sentence meanings.

Pragmatic. This is high-level knowledge which relates to the use of sentences in different contexts and how the context affects the meaning of the sentences.

World. World knowledge relates to the language a user must have in order to understand and carry on a conversation. It must include an understanding of the other person's beliefs and goals.

The approaches taken in developing language understanding programs generally follow the above levels or stages. When a string of words has been detected, the sentences are parsed or analyzed to determine their structure (syntax) and grammatical correctness. The meanings (semantics) of the sentences are then determined and appropriate representation structures created for the inferencing programs. The whole process is a series of transformations from the basic speech sounds to a complete set of internal representation structures.

Understanding written language or text is easier than understanding speech. To understand speech, a program must have all the capabilities of a text understanding program plus the facilities needed to map spoken sounds (often corrupted by noise) into textual form. In this chapter, we focus on the easier problem, that of natural language understanding from textual input and information processing. The process of translating speech into written text is considered in Chapter 13 under Pattern Recognition and the process of generating text is considered later in this chapter.

General Approaches to Natural Language Understanding

Essentially, there have been three different approaches taken in the development of natural language understanding programs, (1) the use of keyword and pattern matching, (2) combined syntactic (structural) and semantic directed analysis, and (3) comparing and matching the input to real world situations (scenario representations).

The keyword and pattern matching approach is the simplest. This approach was first used in programs such as ELIZA described in Chapter 10. It is based on the use of sentence templates which contain key words or phrases such as "______ my mother ______," "I am __________," and, "I don't like ______," that are matched against input sentences. Each input template has associated with it one or more output templates, one of which is used to produce a response to the given input. Appropriate word substitutions are also made from the input to the output to produce the correct person and tense in the response (I and me into you to give replies like "Why are you ______"). The advantage of this approach is that ungrammatical, but meaningful sentences are still accepted. The disadvantage is that no actual knowledge structures are created; so the program does not really understand.

The third approach is based on the use of structures such as the frames or scripts described in Chapter 7. This approach relies more on a mapping of the input to prescribed primitives which are used to build larger knowledge structures. It depends on the use of constraints imposed by context and world knowledge to develop an understanding of the language inputs. Prestored descriptions and details for commonly occurring situations or events are recalled for use in understanding a new situation. The stored events are then used to fill in missing details about the current scenario. We will be returning to this approach later in this chapter. Its advantage is that much of the computation required for syntactical analysis is bypassed. The disadvantage is that a substantial amount of specific, as well as general world knowledge must be prestored.

The second approach is one of the most popular approaches currently being used and is the main topic of the first part of this chapter. With this approach, knowledge structures are constructed during a syntactical and semantical analysis of the input sentences. Parsers are used to analyze individual sentences and to build structures that can be used directly or transformed into the required knowledge formats. The advantage of this approach is in the power and versatility it provides. The disadvantage is the large amount of computation required and the need for still further processing to understand the contextual meanings of more than one sentence.

12.3 GRAMMARS AND LANGUAGES

A language L can be considered as a set of strings of finite or infinite length, where a string is constructed by concatenating basic atomic elements called symbols. The finite set v of symbols of the language is called the alphabet or vocabulary. Among all possible strings that can be generated from v are those that are well-formed, the sentences (such as the sentences found in a language like English). Well-formed sentences are constructed using a set of rules called a grammar. A grammar G is a formal specification of the sentence structures that are allowable in the language, and the language generated by the grammar G is denoted by L(G).

More formally, we define a grammar G as

$$G = (v_n, v_t, s, p)$$

where v_n is a set of nonterminal symbols, v_t a set of terminal symbols, s is a starting symbol, and p is a finite set of productions or rewrite rules. The alphabet v is the union of the disjoint sets v_n and v_t which includes the empty string e. The terminals v_t, are symbols which cannot be decomposed further (such as adjectives, nouns or verbs in English), whereas the nonterminals can be decomposed (such as a noun or verb phrase).

A general production rule from P has the form

$$xyz \rightarrow xwz$$

where x, y, z, and w are strings from v. This rule states that y should be rewritten as w in the context of x to z where x and z can be any string including the empty string e.

As an example of a simple grammar G, we choose one which has component parts or constituents from English with vocabulary Q given by

$$Q_N = \{S, NP, N, VP, V, ART\}$$
$$Q_T = \{boy, popsicle, frog, ate, kissed, flew, the, a\}$$

and rewrite rules given by

$$
\begin{aligned}
P:\quad \mathrm{S} &\rightarrow \mathrm{NP\ VP}\\
\mathrm{NP} &\rightarrow \mathrm{ART\ N}\\
\mathrm{VP} &\rightarrow \mathrm{V\ NP}\\
\mathrm{N} &\rightarrow \text{boy} \mid \text{popsicle} \mid \text{frog}\\
\mathrm{V} &\rightarrow \text{ate} \mid \text{kissed} \mid \text{flew}\\
\mathrm{ART} &\rightarrow \text{the} \mid \text{a}
\end{aligned}
$$

where the vertical bar indicates alternative choices.

S is the initial symbol (for sentence here), NP stands for noun phrase, VP stands for verb phrase, N stands for noun, V is an abbreviation for verb, and ART stands for article.

The grammar G defined above generates only a small fraction of English, but it illustrates the general concepts of generative grammars. With this G, sentences such as the following can be generated.

The boy ate a popsicle.
The frog kissed a boy.
A boy ate the frog.

To generate a sentence, the rules from P are applied sequentially starting with S and proceeding until all nonterminal symbols are eliminated. As an example, the first sentence given above can be generated using the following sequence of rewrite rules:

$$
\begin{aligned}
\mathrm{S} &\rightarrow \mathrm{NP\ VP}\\
&\rightarrow \mathrm{ART\ N\ VP}\\
&\rightarrow \text{the N VP}\\
&\rightarrow \text{the boy VP}\\
&\rightarrow \text{the boy V NP}\\
&\rightarrow \text{the boy ate NP}\\
&\rightarrow \text{the boy ate ART N}\\
&\rightarrow \text{the boy ate a N}\\
&\rightarrow \text{the boy ate a popsicle}
\end{aligned}
$$

It should be clear that a grammar does not guarantee the generation of meaningful sentences, only that they are structurally correct. For example, a gramatically correct, but meaningless sentence like "The popsicle flew a frog" can be generated with this grammar.

We learn a language by learning its structure and not by memorizing all of the sentences we have ever heard, and we are able to use the language in a variety of ways because of this familiarity. Therefore, a useful model of language is one which characterizes the permissible structures through the generating grammars. Unfortunately, it has not been possible to formally characterize natural languages with a simple grammar. In other words, it has not been possible to classify natural languages in a mathematical sense as we did in the example above. More constrained languages (formal programming languages) have been classified and studied through the use of similar grammars, including the Chomsky classes of languages (1965).

The Chomsky Hierarchy of Generative Grammars

Noam Chomsky defined a hierarchy of grammars he called types 0, 1, 2, and 3. Type 0 grammar is the most general. It is obtained by making the simple restriction that y cannot be the empty string in the rewrite form $xyz \rightarrow xwz$. This broad generality requires that a computer having the power of a Turing machine be used to recognize sentences of type 0.

The next level down in generality is obtained with type 1 grammars which are called context-sensitive grammars. They have the added restriction that the length of the string on the right-hand side of the rewrite rule must be at least as long as the string on the left-hand side. Furthermore, in productions of the form $xyz \rightarrow xwz$, y must be a single nonterminal symbol and w, a nonempty string. Typical rewrite rules for type 1 grammars take the forms

$$\begin{aligned} S &\rightarrow aS \\ S &\rightarrow aAB \\ AB &\rightarrow BA \\ aA &\rightarrow ab \\ aA &\rightarrow aa \end{aligned}$$

where the capitalized letters are nonterminals and the lower case letters terminals.

The third type, the type 2 grammar, is known as a context-free grammar. It is characterized by rules with the general form <symbol> → <symbol1> . . . <symbolk> where $k \geq 1$ and where the left-hand side is a single nonterminal symbol, $A \rightarrow xyz$ where A is a single nonterminal. Productions for this type take forms such as

$$\begin{aligned} S &\rightarrow aS \\ S &\rightarrow aSb \\ S &\rightarrow aB \\ S &\rightarrow aAB \\ A &\rightarrow a \\ B &\rightarrow b \end{aligned}$$

The final and most restrictive type is type 3. It is also called a finite state or regular grammar, whose rules are characterized by the forms

$$\begin{aligned} A &\rightarrow aB \\ A &\rightarrow a \end{aligned}$$

The languages generated by these grammars are also termed types 0, 1 (context-sensitive), 2 (context-free), and 4 (regular) corresponding to the grammars which generate them.

The regular and context-free languages have been the most widely studied and best understood. For example, formal programming languages are typically based on context-free languages. Consequently, much of the work in human language understanding has been related to these two types. This is understandable since type 0 grammars are too general to be of much practical use, and type 1 grammars are not that well understood yet.

Structural Representations

It is convenient to represent sentences as a tree or graph to help expose the structure of the constituent parts. For example, the sentence "The boy ate a popsicle" can be represented as the tree structure depicted in Figure 12.1. Such structures are also called phrase markers because they help to mark or identify the phrase structures in a sentence.

The root node of the tree in Figure 12.1 corresponds to the whole sentence S, and the constituent parts of S are subtrees. For this example, the left subtree is a noun phrase, and the right subtree a verb phrase. The leaf or terminal nodes contain the terminal symbols from v_t.

A tree structure such as the above represents a large number of English sentences. It also represents a large class of ill-formed strings that are nonsentences like "The popsicle flew a tree." This satisfies the above structure, but has no meaning.

For purposes of computation, a tree must be represented as a record, a list or similar data structure. We saw in earlier chapters that a list can always be used to represent a tree structure. For example, the tree in Figure 12.1 could be represented as the list

```
(S  (NP  ((ART the)
          (N boy))
    (VP  (V ate)
         (NP  (ART a)
              (N popsicle)))))
```

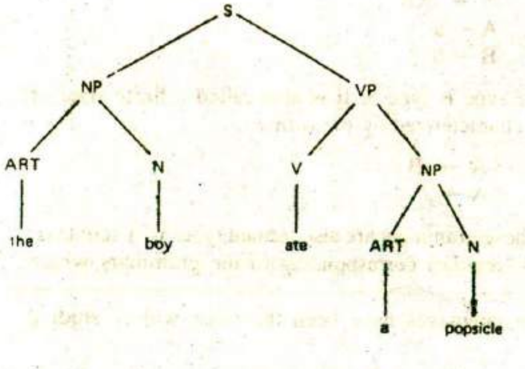


Figure 12.1 A phrase marker or syntactic tree.

A more extensive English grammar than the one given above can be obtained with the addition of other constituents such as prepositional phrases PP, adjectives ADJ, determiners DET, adverbs ADV, auxiliary verbs AUX, and so on. Additional rewrite rules permitting the use of these constituents could include some of the following:

$$\begin{aligned}
\text{PP} &\rightarrow \text{PREP NP}\\
\text{VP} &\rightarrow \text{V ADV}\\
\text{VP} &\rightarrow \text{V PP}\\
\text{VP} &\rightarrow \text{V NP PP}\\
\text{VP} &\rightarrow \text{AUX V NP}\\
\text{DET} &\rightarrow \text{ART ADJ}\\
\text{DET} &\rightarrow \text{ART}
\end{aligned}$$

These extensions broaden the types of sentences that can be generated by permitting the added constituents in sentence forms such as

The mean boy locked the dog in the house.

The cute girl worked to make some extra money.

These sentences have the form S → NP VP PP.

Transformational Grammars

The generative grammars described above generally produce different structures for sentences having different syntactical forms even though they may have the same semantic content. For example, the active and passive forms of a sentence will result in two different phrase marker structures. The sentences "Joe kissed Sue" (active voice) and "Sue was kissed by Joe" (passive voice) result in the structures depicted in Figure 12.2 where the subject and object roles for Joe and Sue are switched.

Obtaining different structures from sentences having the same meaning is undesirable in language understanding systems. Sentences with the same meaning should always map to the same internal knowledge structures. In an attempt to repair these shortcomings in generative grammars, Chomsky (1965) extended them by incorporating two additional components to the basic syntactic component. The added components provide a mechanism to produce single representations for sentences having the same meanings through a series of transformations. This extended grammar is called a transformational generative grammar. Its additions include a semantic component and a phonological component. They are used to interpret the output of the syntactic component by producing meanings and sound sequences. The transformations are essentially tree manipulation rules which depend on the use of an extended lexicon (dictionary) containing a number of semantic features for each word.

Using a transformational generative grammar, a sentence is analyzed in two stages. In one stage the basic structure of the sentence is analyzed to determine the

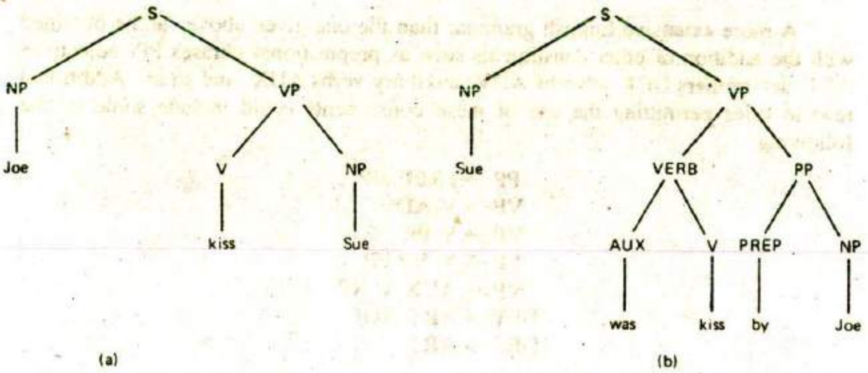


Figure 12.2 Structures for (a) active and (b) passive voice.

grammatical constituent parts. This reveals the surface structure of the sentence, the way the sentence is used in speech or in writing. This structure can be transformed into another one where the deeper semantic structure of the sentence is determined.

Application of the transformation rules can produce a change from passive voice to active voice, change a question to declarative form, and handle negations, subject-verb agreement, and so on. For example, the structure in 12.2(b) could be transformed to give the same basic structure as that of 12.2(a) as is illustrated in Figure 12.3.

Transformational grammars were never widely adopted as computational models of natural language. Instead, other grammars, including case grammars, have had more influence on such models.

Case Grammars

A case relates to the semantic role that a noun phrase plays with respect to verbs and adjectives. Case grammars use the functional relationships between noun phrases and verbs to reveal the deeper case of a sentence. These grammars use the fact

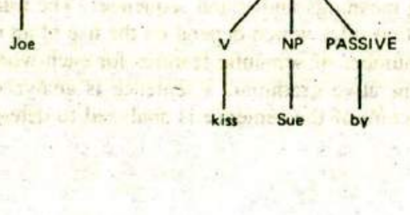


Figure 12.3 Passive voice transformed to active voice.

that verbal elements provide the main source of structure in a sentence since they describe the subject and objects.

In inflected languages like Latin, nouns generally have different ending forms for different cases. In English these distinctions are less pronounced and the forms remain more constant for different cases. Even so, they provide some constraints. English cases are the nominative (subject of the verb), possessive (showing possession or ownership), and objective (direct and indirect objects). Fillmore (1968, 1977) revived the notion of using case to extract the meanings of sentences. He extended the transformational grammars of Chomsky by focusing more on the semantic aspects of a sentence.

In case grammars, a sentence is defined as being composed of a proposition P, a tenseless set of relationships among verbs and noun phrases and a modality constituent M, composed of mood, tense, aspect, negation, and so on. Thus, a sentence can be represented as

$$S \rightarrow M + P$$

where P in turn consists of one or more distinct cases C1, C2, . . . , Ck,

$$P \rightarrow C1 + C2 + \ldots + Ck.$$

The number of cases suggested by Fillmore were relatively few. For example, the original list contained only some six cases. They relate to the actions performed by agents, the location and direction of actions, and so on. For example, the case of an instigator of an action is the agentive (or agent), the case of an instrument or object used in an action is the instrumental, and the case of the object receiving the action or change is the objective. Thus, in sentences like "The soldier struck the suspect with the rifle butt" the soldier is the agentive case, the suspect the objective case, and the rifle butt the instrumental case. Other basic cases include dative (an animate entity affected by an action), factitive (the case of the object or of being that which results from an event), and locative (the case of location of the event). Additional cases or substitutes for those given above have since been introduced, including beneficiary, source, destination, to or from, goal, and time.

Case frames are provided for verbs to identify allowable cases. They give the relationships which are required and those which are optional. For the above sentence, a case frame for the verb struck might be

STRUCK[OBJECTIVE (AGENTIVE) (INSTRUMENTAL)]

This may be interpreted as stating that the verb struck must occur in sentences with a noun phrase in the objective case and optionally (parentheses indicate optional use) with noun phrases in the agentive and instrumental cases.

A tree representation for a case grammar will identify the words by their modality and case. For example, a case grammar tree for the sentence "Sue did not take the car" is illustrated in Figure 12.4.

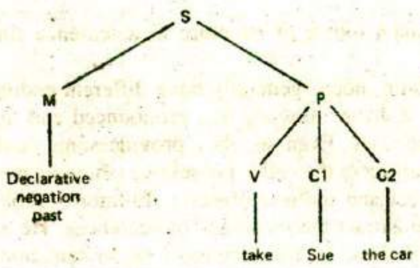


Figure 12.4 Case grammar tree representation.

To build a tree structure like this requires that a word lexicon with sufficient information be available in which to determine the case of sentence elements.

Systemic Grammars

Systemic grammars emphasize function and purpose in the analysis of language. They attempt to account for the personal and social aspects which influence communication through language. As such, context and pragmatics play a more important role in systemic grammars.

Systemic grammars were introduced by Michael Halliday (1961) and Winograd (1972) in an attempt to account for the principles that underlie the organization of different structures. He classifies language by three functions which relate to content, purpose, and coherence.

1. The *ideational function* relates to the content intended by the speaker. This function provides information about the kinds of activities being described, who the actors are, whether there are other participants, and the circumstances related to time and place. These concepts bear some similarity to the case grammars described above.

2. The *interpersonal function* is concerned with the purpose and mood of the statements, whether a question is being asked, an answer being given, a request being made, an opinion being offered, or information given.

3. The *textual function* dictates the necessity for continuity and coherence between the current and previously stated expressions. This function is concerned with the theme of the conversation, what is known, and what is newly expressed.

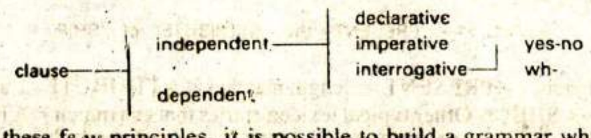
Halliday proposed a model of language which consisted of four basic categories.

Language units. A hierarchy for sentences based on the sentence, clause, phrase group, word, and morpheme.

Role structure of units. A unit consists of one or more units of lower rank based on its role, such as subject, predicate, complement, or adjunct.

Classification of units. Units are classified by the role they play at the next higher level. For example, the verbal serves as the predicate, the nominal serves as the subject or complement, and so on.

System constraints. These are constraints in combining component features. For example, the network structure given below depicts the constraints in an interpretation.



Given these few principles, it is possible to build a grammar which combines much semantic information with the syntactic. Many of the ideas from systemic grammars were used in the successful system SHRDLU developed by Terry Winograd (1972). This system is described later in the chapter.

Semantic Grammars

Semantic grammars encode semantic information into a syntactic grammar. They use context-free rewrite rules with nonterminal semantic constituents. The constituents are categories, or metasymbols such as attribute, object, present (as in display), and ship, rather than NP, VP, N, V, and so on. This approach greatly restricts the range of sentences which can be generated and requires a large number of rewrite rules.

Semantic grammars have proven to be successful in limited applications including LIFER, a data base query system distributed by the Navy which is accessible through ARPANET (Hendrix et al., 1978), and a tutorial system named SOPHIE which is used to teach the debugging of circuit faults. Rewrite rules in these systems essentially take the forms

S → What is <OUTPUT-PROPERTY> of <CIRCUIT-PART>?
OUTPUT-PROPERTY → the <OUTPUT-PROP>
OUTPUT-PROPERTY → < OUTPUT-PROP>
CIRCUIT-PART → C23
CIRCUIT-PART → D12
OUTPUT-PROP → voltage
OUTPUT-PROP → current

In the LIFER system, there are rules to handle numerous forms of wh-queries such as

What is the name and location of the carrier nearest to New York
Who commands the Kennedy

Which convoy escorts have inoperative radar units

When will they be repaired

What Soviet ship has hull number 820

These sentences are analyzed and words matched to metasymbols contained in lexicon entries. For example, the input statement "Print the length of the Enterprise" would fit with the LIFER top grammar rule (L.T.G.) of the form

<L.T.G.> → <PRESENT> the <ATTRIBUTE> of <SHIP>

where print matches <PRESENT>, length matches <ATTRIBUTE>, and the Enterprise matches <SHIP>. Other typical lexicon entries that can match <ATTRIBUTE> include CLASS, COMMANDER, FUEL, TYPE, BEAM, LENGTH, and so on.

LIFER can also accommodate elliptical (incomplete) inputs. Given the query "What is the length of the Kennedy?" a subsequent query consisting of the abbreviated form "of the Enterprise?" will elicit a proper response (see also the third and fourth example queries above).

Semantic grammars are suitable for use in systems with restricted grammars since computation is limited. They become unwieldy when used with general purpose language understanding systems, however.

12.4 BASIC PARSING TECHNIQUES

Before the meaning of a sentence can be determined, the meanings of its constituent parts must be established. This requires a knowledge of the structure of the sentence, the meanings of individual words and how the words modify each other. The process of determining the syntactical structure of a sentence is known as parsing.

Parsing is the process of analyzing a sentence by taking it apart word-by-word and determining its structure from its constituent parts and subparts. The structure of a sentence can be represented with a syntactic tree or a list as described in the previous section. The parsing process is basically the inverse of the sentence generation process since it involves finding a grammatical sentence structure from an input string. When given an input string, the lexical parts or terms (root words) must first be identified by type, and then the role they play in a sentence must be determined. These parts can then be combined successively into larger units until a complete tree structure has been completed.

To determine the meaning of a word, a parser must have access to a lexicon. When the parser selects a word from the input stream it locates the word in the lexicon and obtains the word's possible function and other features, including semantic information. This information is then used in building a tree or other representation structure. The general parsing process is illustrated in Figure 12.5.

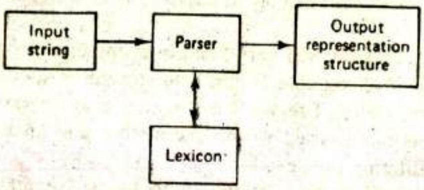


Figure 12.5 Parsing an input to create an output structure.

The Lexicon

A lexicon is a dictionary of words (usually morphemes or root words together with their derivatives), where each word contains some syntactic, semantic, and possibly some pragmatic information. The information in the lexicon is needed to help determine the function and meanings of the words in a sentence. Each entry in a lexicon will contain a root word called the head. Different derivatives of the word, if any, will also be given, and the roles it can play in a sentence (e.g. its part of speech and sense). A fragment of a simplified lexicon is illustrated in Figure 12.6.

The abbreviations 1s, 2s, . . . , 3p in Figure 12.6 stand for first person singular, second person singular, . . . , third person plural, respectively. Note that some words have more than one type such as can which is both a noun and a verb, and orange which is both an adjective and a noun. A lexicon may also be organized to contain separate entries for words with more than one function by giving them separate identities, can1 and can2. Alternatively, the entries in a lexicon could be grouped and given by word category (by articles, nouns, pronouns, verbs,

Word	Type	Features
a	Determiner	{3s}
be	Verb	Trans: intransitive
boy	Noun	{3s}
can	Noun	{1s, 2s, 3s, 1p, 2p, 3p}
	Verb	Trans: intransitive
carried	Verb	Form: past, past participle
.	.	.
.	.	.
orange	Adjective	
	Noun	{3s}
the	Determiner	{3s, 3p}
to	Preposition	
we	Pronoun	{1p}
		Case: subjective
yellow	Adjective	

Figure 12.6 Typical entries in a lexicon

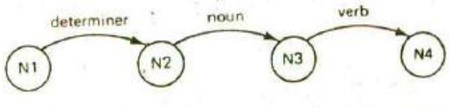
and so on), and all words contained within the lexicon listed within the categories to which they belong.

The organization and entries of a lexicon will vary from one implementation to another, but they are usually made up of variable length data structures such as lists or records arranged in alphabetical order. The word order may also be given in terms of usage frequency so that frequently used words like a, the, and an will appear at the beginning of the list facilitating the search.

Access to the words may be facilitated by indexing, with binary searches, hashing, or combinations of these methods. A lexicon may also be partitioned to contain a base lexicon set of general, frequently used words and domain specific components of words.

Transition Networks

Transition networks are another popular method used to represent formal and natural language structures. They are based on the application of directed graphs (digraphs) and finite state automata. A transition network consists of a number of nodes and labeled arcs. The nodes represent different states in traversing a sentence, and the arcs represent rules or test conditions required to make the transition from one state to the next. A path through a transition network corresponds to a permissible sequence of word types for a given grammar. Thus, if a transition network can be successfully traversed, it will have recognized a permissible sentence structure. For example, a network used to recognize a sentence consisting of a determiner, a noun and a verb ("The child runs") would be represented by the three-node graph as follows.



Starting at node N1, the transition from node N1 to N2 will be made if a determiner is the first input word found. If successful, state N2 is entered. The transition from N2 to N3 can then be made if a noun is found next. The final transition (from N3 to N4) will be made if the last word is a verb. If the three-word category sequence is not found, the parse fails. Clearly, this type of network is very limited since it will only recognize simple sentences of the form DET N V.

The utility of a network such as this could be increased if more than a single choice were permitted at some of the nodes. For example, if several arcs were constructed between nodes N1 and N2 where each arc represented a different noun phrase, the number of permissible sentence types would be increased substantially. Individual arcs could be a noun, a pronoun, a determiner followed by a noun, a determiner followed by an adjective followed by a noun, or some other type of noun phrase which we wish the parser to be capable of recognizing. These alternatives are depicted in Figure 12.7.

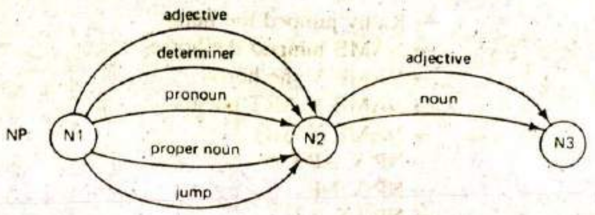


Figure 12.7 A noun phrase segment of a transition network.

To move from state N1 to N2 in this transition network, it is necessray to first find an adjective, a determiner, a pronoun, a proper noun, or none of these by "jumping" directly to N2. This network extends the possible types of sentences that can be recognized substantially over the simple network given above. For example, it will recognize noun phrases having forms such as

Big white fluffy clouds
Our bright children
A large beautiful white flower
Large green leaves
Buildings
Boston's best seafood restaurants

Top-Down versus Botton-up Parsing

Parsers may be designed to process a sentence using either a top-down or a bottom-up approach. A top-down parser begins by hypothesizing a sentence (the symbol S) and successively predicting lower level constituents until individual preterminal symbols are written. These are then replaced by the input sentence words which match the terminal categories. For example, a possible top-down parse of the sentence "Kathy jumped the horse" would be given by

$$\begin{aligned}
S &\rightarrow \text{NP VP}\\
&\rightarrow \text{NAME VP}\\
&\rightarrow \text{Kathy VP}\\
&\rightarrow \text{Kathy V NP}\\
&\rightarrow \text{Kathy jumped NP}\\
&\rightarrow \text{Kathy jumped ART N}\\
&\rightarrow \text{Kathy jumped the N}\\
&\rightarrow \text{Kathy jumped the horse}
\end{aligned}$$

A bottom-up parse, on the other hand, begins with the actual words appearing in the sentence and is, therefore, data driven. A possible bottom-up parse of the same sentence might proceed as follows.

→ Kathy jumped the horse
→ NAME jumped the horse
→ NAME V the horse
→ NAME V ART horse
→ NAME V ART N
→ NP V ART N
→ NP V NP
→ NP VP
→ S

Words in the input sentence are replaced with their syntactic categories and those in turn are replaced by constitutents of the same or smaller size until S has been rewritten or until failure occurs.

Deterministic versus Nondeterministic Parsers

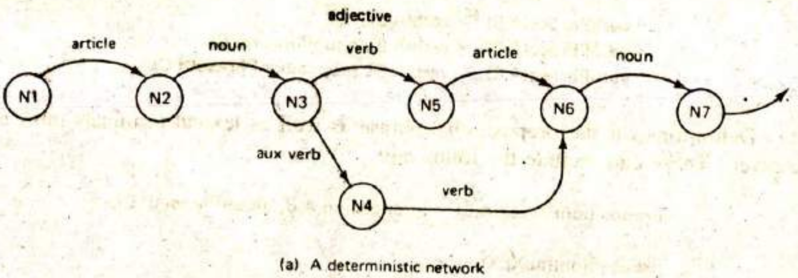
Parsers may also be classified as deterministic or nondeterministic depending on the parsing strategy employed. A deterministic parser permits only one choice (arc) for each word category. Thus, each arc will have a different test condition. Consequently, if an incorrect test choice is accepted from some state, the parse will fail since the parser cannot backtrack to an alternative choice. This may occur, for example, when a word satisfies more than one category such as a noun and a verb or an adjective, noun, and verb. Clearly, in deterministic parsers, care must be taken to make correct test choices at each stage of the parsing. This can be facilitated with a look-ahead feature which checks the categories of one or more subsequent words in the input sentence before deciding in which category to place the current word. Some researchers prefer to use deterministic parsing since they feel it more closely models the way humans parse input sentences.

Nondeterministic parsers permit different arcs to be labeled with the same test. Consequently, the next test from any given state may not be uniquely determined by the state and the current input word. The parser must guess at the proper constituent and then backtrack if the guess is later proven to be wrong. This will require saving more than one potential structure during parts of the network traversal. Examples of both deterministic and nondeterministic parsers are presented in Figure 12.8.

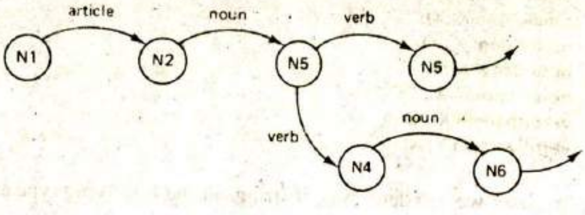
Suppose the following sentence is given to a deterministic parser with the grammar given by the network of Figure 12.8(a): "The strong bear the loads." If the parser chose to recognize strong as an adjective and bear as a noun, the parse would fail, since there is no verb following bear. A nondeterministic parser, on the other hand, would simply recover by backtracking when failure was detected and then taking another arc which accepted strong as a noun.

Example of a Simple Parser in Prolog

The reader may have noticed the close similarity between rewrite rules and Horn clauses, especially when the Horn clauses are written in the form of PROLOG



(a) A deterministic network



(b) A nondeterministic network

Figure 12.8 Deterministic and nondeterministic networks.

rules. This similarity makes it a straightforward task to write parsers in a language like PROLOG. For example, the grammar rule that states that S is a sentence if it is a noun phrase followed by a verb phrase (S → NP VP) may be written in PROLOG as

```
sentence(A,C) :- nounPhrase(A,B), verbPhrase(B,C)
```

The variables A, B, and C in this statement represent lists of words. The argument A is the whole list of words to be tested as a sentence, and C is the list of remaining words, if any. Similar assumptions hold for A, B, and C in the noun and verb phrase conditions respectively.

Rule definitions which rewrite the noun phrases and verb phrases must also be defined. Thus, an NP may be defined with statements such as the following:

```
nounPhrase(A,C) :=article(A,B), noun(B,C).
nounPhrase(A,B) :=noun(A,B).
```

Like the above rule, these rules state that (1) a noun phrase can be either an article which consists of a list A and remaining list B (if any) and a noun which is a list B and remaining list C or (2) a noun consisting of the list A with remaining list B (if any). Similarly, a verb phrase may be defined with rules like the following:

```
verbPhrase(A,B) := verb(A,B).
verbPhrase(A,C) := verb(A,B), nounPhrase(B,C).
verbPhrase(A,C) := verb(A,B), prepositionPhrase(B,C).
```

Definitions for the prepositional phrase as well as lexical terminals must also be given. These can include the following:

```
prepositionPhrase(A,C) := preposition(A,B), nounPhrase(B,C).

preposition([at|X],X).
article([a|X],X).
article([the|X],X).
noun([dog| X],X).
noun([cow|X],X).
noun([moon| X],X).
verb([barked|X],X).
verb([winked|X],X).
```

With this simple parser we can determine if strings of the following type are grammatically correct.

The dog barked at the cow.
The moon winked at the dog.
A cow barked at a moon.

To do so, we must enter sentence queries as lists such as the following for the PROLOG interpreter:

```
?- sentence([the,dog,barked,at,the,moon],X]).
X = [ ]
?- sentence([barked,a,moon,dog,the],X].
no
```

Since the remainder of the sentence bound to X is the empty set, it is recognized as correct. The second sentence failed since it could not instantiate with the correct constituent parts.

Of course, for a parser to be of much practical use, other constituents and a great many more words should be defined. The example illustrates the utility of using PROLOG as a basic parser.

Recursive Transition Networks

The simple networks described above are not powerful enough to recognize the variety of sentences a human language system could be expected to cope with. In fact, they fail to recognize all languages that can be generated by a context-free grammar. Other extensions are needed to accept a wider range of sentences but still avoid the necessity for large complex networks. We can achieve such extensions by labeling some arcs as a separate network state (such as an NP) and then constructing a subnetwork which recognizes the different noun phrases required. In this way, a single subnetwork for an NP can be called from several places in a sentence. Similar arcs can be labeled for other sentence constituents including VP, PP (prepositional phrases) and others. With these additions, complex sentences having embedded phrases can be parsed with relatively simple networks. This leads directly to the notion of using recursion in a network.

A recursive transition network (RTN) is a transition network which permits arc labels to refer to other networks (including the network's own name), and they in turn may refer back to the referring network rather than just permitting word categories used previously. For example, an RTN described by William Woods (1970) is illustrated in Figure 12.9 where the main network calls two subnetworks and an NP and PP network as illustrated in 12.9(b) and (c).

The top network in the figure is the top level (sentence) network, and the lower level networks are for NP and PP arc states. The arcs corresponding to these states will be traversed only if the corresponding subnetworks (b) or (c) are successfully traversed.

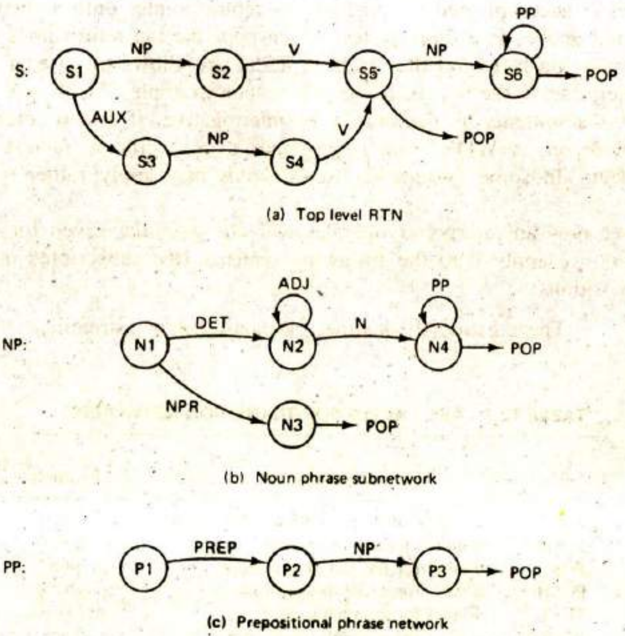


Figure 12.9 Recursive transition network.

In traversing a network, it is customary to test the arcs in a clockwise order. Thus, in the top level RTN, the NP arc will be called first. If this arc fails, the arc labeled AUX will be tested next.

During the traversal of an RTN, a record must be maintained of the word position in the input sentence and the current state or node position and return nodes to be used as return points when control has been transferred to a lower level network. This information can be maintained as a triple like POS CND RLIST where POS is the current input word position, CND is the current node, and RLIST is the list of return points. The RLIST can be maintained as a stack data structure.

In Figure 12.9, the arc named POP is used as a dummy arc to signal the successful completion of the subnetwork and a return to the node following the arc from which it was called. Some other arc types that will be useful in what follows are summarized in Table 12.1.

The CAT arc represents a test for a specific word category such as a noun or a verb. When the input word is of the specified category, the CAT test succeeds and the input pointer is advanced to the next word. The JUMP arc may be traversed without satisfying any test condition in which case the word pointer is not advanced when a JUMP arc is traversed. An arc labeled with a state, such as NP or PP, is defined as a PUSH arc. This arc initiates a call to another network with the indicated state (such as an NP or PP). When a PUSH arc is taken, a return state must be saved. This is accomplished by pushing the return pointer onto a stack. The POP arc, as noted above, is a dummy test which pops the top return node pointer that was previously pushed onto the stack. A TEST arc allows the use of an arbitrary test to determine if the arc is to be taken. For example, TEST can be used to determine if a sentence is declarative or interrogative, if one or more negatives occur, and so on. A WORD arc corresponds to a specific word test such as to, from, and at. (In some systems a list of words may apply rather than a single word.)

To see how an interpreter operates with the grammar given for the RTN of Figure 12.6, we apply it to the following sentence (the subscripted numbers give the word positions):

$$_1\text{The}_2\text{big}_3\text{tree}_4\text{shades}_5\text{tne}_6\ \text{old}_7\text{house}_8\text{by}_9\text{the}_{10}\text{stream}_{11}.$$

TABLE 12.1 ARC LABELS FOR TRANSITION NETWORKS

Type of arc	Purpose of arc	Example
CAT	a test label for the current word category	V
JUMP	requires no test to succeed	jump
POP	a label for the end of a network	pop
PUSH	a label for a call to a network	NP
TEST	a label for an arbitrary test	negatives
WORD	a label for a specific word type	from

Starting with CND set to S1, POS set to 1, and RLIST set to nil, the first arc test (NP) would be completed. Since this test is for a state, the parser would PUSH the return node S2 onto RLIST, set CND to N1, and call the NP network. Trying the first test DET (a CAT test) in the NP network, a match would be found with word position 1. This would result in CND being updated to N2 and POS to position 2. The next word (big) satisfies the ADJ test causing CND to be updated to N2 again, and POS to be updated to position 3. The ADJ test is then repeated for the word tree, but it fails. Hence, the arc test for N is made next with no change made to POS and CND. This time the test succeeds resulting in updates of N4 to CND and position 4 to POS. The next test is the POP which signals a successful completion of the NP network and causes the return node (S1) to be retrieved from the RLIST stack and CND to be updated with S2. POP does not cause an advance in the word position POS.

The only possible test from S2 is for category V which succeeds on the word "shades" with resultant updates of S5 to CND and 5 to POS. At S5, the only possible test is the NP. This again invokes a call to the lower level NP network which is traversed successfully with the noun phrase "the old house." After a return to the main network, CND is set to S6 and POS is set to position 6. At this point, the lower PP network is called with CND being set to P1 and S6 pushed onto RLIST. From P1, the CAT test for PREP passes with CND being set to P2 and POS being set to 9. NP is then called with CND being set to N1 and P2 being pushed onto RLIST. As before, the NP network is traversed with the noun phrase "the stream" resulting in a POS value of 11, P3 being popped from RLIST and a return to that node. The test at P3 (POP) results in S6 being popped from RLIST and a return to the S6 node. Finally, the POP test at N6, together with the period at position 11 results in a successful traversal and acceptance of the sentence.

During a network traversal, a parse can fail if (1) the end of the input sentence (a period) has been reached when the test from the CND node value is not a terminal (POP) value or (2) if a word in the input sentence fails to satisfy any of the available arc tests from some node in the network.

The number of sentences accepted by an RTN can be extended if backtracking is permitted when a failure occurs. This requires that states having alternative transitions be remembered until the parse progresses past possible failure points. In this way, if a failure occurs at some point, the interpreter can backtrack and try alternative paths. The disadvantage with this approach is that parts of a sentence may be parsed more than one time resulting in excessive computations.

Augmented Transition Networks

The networks considered so far are not very useful for language understanding. They have only been capable of accepting or rejecting a sentence based on the grammar and syntax of the sentence. To be more useful, an interpreter must be able to build structures which will ultimately be used to create the required knowledge entities for an AI system. Furthermore, the resulting data structures should contain more information than just the syntactic information dictated by the grammar alone. Semantic information should also be included. For example, a number of sentence features can also be established and recorded, such as the subject NP, the object NP, the subject-verb number agreement, the mood (declarative or interrogative), tense, and so on. This means that additional tests must be performed to determine the possible semantics a sentence may have. Without these additional tests, much ambiguity will still be present and incorrect or meaningless sentences accepted.

We can achieve the additional capabilities required by augmenting an RTN with the ability to perform additional tests and store immediate results as a sentence is being parsed. When an RTN is given these additional features, it is called an augmented transition network or ATN.

When building a representation structure, an ATN uses a number of different registers as temporary storage to hold the different sentence constituents. Thus, one set of registers would be used for an NP network, one for a PP network, one for a V, and so on. Using the register contents, an ATN builds a partial structural description of the sentence as it moves from state to state in the network. These registers provide temporary storage which is easily modified, switched, or discarded until the final sentence structure is constructed. The registers also hold flags and other indicators used in conjunction with some arcs. When a partial structure has been stored in registers and a failure occurs, the interpreter can clear the registers, backtrack, and start a new set of tests. At the end of a successful parse, the contents of the registers are combined to form the final sentence data structure required for output.

An ATN Specification Language

A specification language developed by Woods (1970, 1986) for ATNs takes the form of an extended context-free grammar. This language is given in Figure 12.10 where the vertical bar indicates alternative choices for a construction and the * (Kleene star) signifies repeatable (zero or more) elements. All nonterminals are enclosed in angle brackets. Some of the capitalized words appearing in the language were defined earlier as arc tests and actions. The other words in uppercase correspond to functions which perform many of the tasks related to the construction of the structure using the registers.

The specification language is read the same as rewrite rules. Thus, it specifies that a transition network is composed of a list of arc sets, where each arc set is in turn a list with first element being a state name and the remaining elements being arcs which emanate from that state. An arc can be any of the forms CAT, JUMP, PUSH, TEST, WORD or POP. For example, as noted earlier, the TEST arc corresponds to an arbitrary test which determines whether the arc is to be traversed or not. Note that a sequence of actions is associated with the arc tests. These actions are executed during the arc traversals. They are used to build pieces of structures such as a tree or a list. The terminal action of any arc specifies the state to which control is passed to complete the transition.

```
<transition net> → (<arc set><arc set>*)
<arc set> → (<state><arc>*)
<arc> → (CAT <category name><test><action>*<term act>)|
          (PUSH <state><test><action>*<term act>)|
          (TST (arbitrary label><test><action>*<term act>)|
          (POP <form><test>)
<action> → (SETR <register><form>)|
             (SENDR <register><form>)|
             (LIFTR <register><form>)
<term act> → (TO <state>)|
               (JUMP <state>)
<form> → (GETR <register>)|
           @|
           (GETF <feature>)|
           (BUILDQ <fragment><register>*)|
           (LIST <form>*)|
           (APPEND <form><form>)|
           (QUOTE <arbitrary structure>)
```

Figure 12.10 A specification language for ATNs.

Among other things, an action can be any of the three function forms SETR, SENDR, and LIFTR which cause the indicated register values to be set to the value of form. Terminal actions can be either TO or JUMP where TO requires that the input sentence pointer should be advanced, and JUMP requires that the pointer remain fixed and the input word continue to be scanned. Finally, a construction form can be any of the seven alternatives in the bottom group of Figure 12.10, including the symbol @ which is a terminal symbol placeholder for form.

The function SETR causes the contents of the indicated registers to be set equal to the value of the corresponding form. This is done at the current level in the network, while SENDR causes it to be done by sending it to the next lower level of computation. LIFTR returns information to the next higher level of computation. The function GETR returns the value of the indicated register, and GETF returns the value of a specified feature for the current input word. As noted before, the value of @ is usually an input word. The function BUILDQ takes lists from the indicated registers (which represent fragments of a parse tree with marked nodes) and builds the sentence structures.

An ATN network similar to the RTN illustrated in Figure 12.9 is presented in Figure 12.11. Note that the arcs in this network have some of the tests described above. These tests will have the basic forms given in Figure 12.10, together with the indicated actions. The actions include building the final sentence structure which may contain more features than those considered thus far, as well as certain semantic features.

Using the specification language, we can represent this particular network with the constituent abbreviations and functions described above in the form of a LISP program. For example, a partial description of the network is depicted in

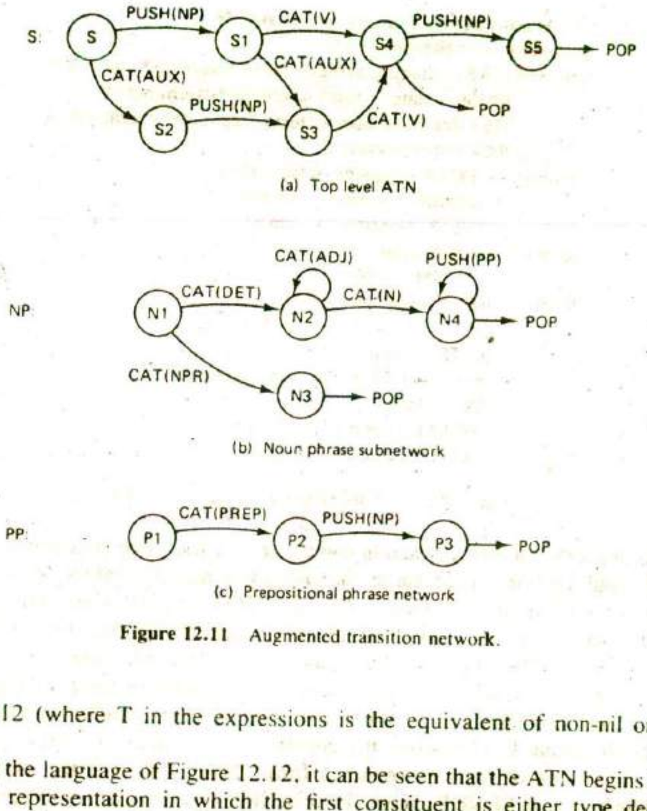


Figure 12.11 Augmented transition network.

Figure 12.12 (where T in the expressions is the equivalent of non-nil or true in LISP).

From the language of Figure 12.12, it can be seen that the ATN begins building a sentence representation in which the first constituent is either type declarative (DCL) or type interrogative (Q for question), depending on whether the first successful test is an NP or AUX, respectively. The next constituent is the subject (SUBJ) NP, and the third is either an auxiliary verb (AUX) or nil. The fourth constituent is a VP. An ATN is traversed much the same as the RTN described above.

An example of its operation will help to demonstrate how the structure is built during a parse of the sentence

"The big dog likes the small boy."

1. Starting with state S, PUSH down a level to the NP network. If an NP is found (T for true), execute lines 2, 3, and 4.

2. In the lower level NP network, the noun phrase "the big dog" is found with successive CAT tests for determiner, adjective, and noun. During these tests, NP registers are set to indicate the word constituents. When the terminal node

```
 1. (S/ (PUSH NP/ T
 2.       (SETR SUBJ @ )
 3.       (SETR TYPE (QUOTE DCL))
 4.       (TO S1)
 5.     (CAT AUX T
 6.       (SETR AUX @ )
 7.       (SETR TYPE (QUOTE Q))
 8.       (TO S2)))
 9. (S1 (CAT V T
10.       (SETR AUX NIL)
11.       (SETR V @ )
12.       (TO S4)))
13.     (CAT AUX T
14.       (SETR AUX @ )
15.       (TO S3)))
16. (S2 (PUSH NP/ T
17.       (SETR SUBJ @ )
18.       (TO S3)))
19. (S3 (CAT V T
20.       (SETR V @ )
21.       (TO S4)))
22. (S4 (POP BUILDQ (S+ + +(VP+)) TYPE SUBJ AUX V) T)
23.     (PUSH NP/ T
24.       (SETR VP BUILDQ (VP (V+) @ ) V))
25.       (TO S5)))
26. (S5 (POP (BUILDQ (S+ + + +) TYPE SUBJ AUX VP) T)
27.     (PUSH PP/ T
28.       (SETR VP (APPEND (GETR VP) (LIST @ )))
29.       (TO S5)))
    . . .
    . . .
```

Figure 12.12 An ATN specification language.

(N4) is tested and the PP test subsequently fails, POP is executed and a return of control is made to statement 2.

3. The register SUBJ is set to the value of @ which is the list structure (NP (dog (big) DEF)) returned from the NP registers. DEF signifies that the determiner is definite.

4. In line 3, register TYPE is set to DCL (for declarative).

5. Control is transferred to S1 with the statement TO in line 4 and the input pointer is moved past the noun phrase to the verb "likes."

6. If an auxiliary verb had been found at the beginning of the sentence instead of an NP, control would have been passed to line 5 where statements 5, 7, and 8 would have been executed. This would have resulted in registers AUX and TYPE being set to the values @ and Q respectively.

7. At S1, a category test is made for a V. Since this succeeds (is T), statements 11, 12, and 13 are executed. This results in register AUX being set to nil, and register V being set to the contents of @, to give (V likes). Control is then passed to S4 and the input pointer is moved to the word "the."

8. If the test for V had failed, and an auxiliary verb had been found, statements 14 and 15 would have been executed.

9. Since S4 is a terminal node, a sentence structure can be built there. This will be the case if the end of the sentence has been reached. If so, the BUILDQ function creates a list structure with first element S, followed by the values of the three registers TYPE, SUBJ, AUX, corresponding to the three plus (+) signs. These are then followed with VP and the contents of the V register. For example, with an input sentence of,

The boy can whistle.

the structure (S DCL (NP (boy) DEF) (AUX can) (VP whistle)) would be constructed from the four registers TYPE, SUBJ, AUX, and V.

10. Because more input words remain, the BUILDQ in line 22 is not executed, and control drops to the next line where a push is made to the lower NP network. As before, the NP succeeds with the structure (NP (boy (small) DEF)) being returned as the value of @. Register VP is then set to the list returned by BUILDQ (line 24) which consists of VP followed by the verb phrase and control is passed to S5.

11. Since S5 is a terminal node and the end of the input sentence has been reached, BUILDQ will build the final sentence structure from the TYPE, SUBJ, AUX, and VP register contents. The final structure constructed is

```
(S DCL (NP (dog (big) DEF))
       (VP (V likes)(NP (boy (small) DEF))))
```

The use of recursion, arc tests, and a variety of arc and node combinations give the ATNs the power of a Turing Machine. This means that an ATN can recognize any language that a general purpose computer can recognize. This versatility also makes it possible to build deep sentence structures rather than just structures with surface features only. (Recall that surface features relate to the form of words, phrases, and sentences, whereas deep features relate to the content or meaning of these elements). The ability to build deep structures requires that other appropriate tests be included to check pronoun references, tense, number agreement, and other features.

Because of their power and versatility, ATNs have become popular as a model for general purpose parsers. They have been used successfully in a number of natural language systems as well as front ends for databases and expert systems.

12.5 SEMANTIC ANALYSIS AND REPRESENTATION STRUCTURES

We have now seen how the structure of a complex sentence can be determined and how a representation of that structure can be constructed using different types of parsers. In particular, it should now be clear how an ATN can be used to build structures for different grammars, like those described in Section 12.3. But, we have not yet explained how the final semantic knowledge structures are created to satisfy the requirements of a knowledge base used to represent some particular world model. Experience has shown the semantic interpretation to be the most difficult stage in the transformation process.

As an example of some of the difficulties encountered in extracting the full intended meaning of some utterances, consider the following situation.

> It turned into a black day. In his haste to catch the flight, he backed over Tom's bicycle. He should never have left it there. It was damaged beyond repair. That caused the tailpipe to break. It would be impossible to make it now. . . . . It was all because of that late movie. He would be heartbroken when he found out about it.

Although a car was never explicitly mentioned, it must be assumed that a car was the object which was backed over Tom's bicycle. A program must be able to infer this. The "black day" metaphor also requires some inference. Days are not usually referred to by color. And sorting out the pronoun references can also be an onerous task for a program. Of the seven uses of it, two refer to the bicycle, two to the flight, two refer to the situation in general, and one to the status of the day. There are also four uses of he referring to two different people and a that which refers to the accident in general. The placement of the pronouns is almost at random making it difficult to give any rule of association. Words that point back or refer to people, places, objects, events, times, and so on that occurred before, are called anaphors. Their interpretation may require the use of heuristics, syntactic and semantic constraints, inference, and other forms of object analysis within the discourse content.

This example should demonstrate again that language cannot be separated from intelligence and reasoning. To fully understand the above situation requires that a program be able to reason about people's goals, beliefs, motives, and facts about the world in general.

The semantic structures constructed from utterances such as the above, must account for all aspects of meaning in what is known as the domain, context, and the task. The *domain* refers to the knowledge that is part of the world model the system knows about. This includes object descriptions, relationships, and other relevant concepts. The *context* relates to previous expressions, the setting and time of the utterances, and the beliefs, desires, and intentions of the speakers. A *task* is part of the service the system offers, such as retrieving information from a data base, providing expert advice, or performing a language translation. The domain, context, and task are what we have loosely referred to before as semantics, pragmatics, and world knowledge.

Semantic interpretations require that utterances be transformed into coherent expressions in the form of FOPL clauses, associative networks, frames, or script-like structures that can be manipulated by the understanding program. There are a number of different approaches to the transformation problem. The approach we have been following up to this point is one in which the transformation is made in stages. In the first stage, a syntactic analysis is performed and a tree-like structure is produced using a parser. This stage is followed by use of a semantic analyzer to produce either an intermediate or final semantic structure.

Another approach is to transform the sentences directly into the target structures with little syntactical analysis. Such approaches typically depend on the use of constraints given by semantic grammars or place strong reliance on the use of key words to extract meaning.

Between these two extremes, are approaches which perform syntactic and semantic analyses concurrently, using semantic information to guide the parse, and the structure learned through the syntactical analysis is used to help determine meaning.

Another dimension related to semantic interpretation is the approach taken in extracting the meaning of an expression, (1) whether it can or should be derived by paraphrasing input utterances and transforming them into structures containing a few generic terms or (2) whether meanings are best derived by composing the meanings of clauses and larger units from the meanings of their constituent parts. These two methods of approach are closely tied to the form of the target semantic structures.

The first of these approaches we call unit or *lexical semantics* to emphasize the role played by the special primitive words used to represent the meanings of all expressions. In this approach the meaning is constructed through a restatement of the expression in terms of linked primitive generic words such as those used in Shank's conceptual dependency theory (Chapter 7).

The second approach is called *compositional semantics* since the meaning of an expression is derived from the meanings ascribed to the constituent parts. The structures created through this approach are usually characterized as logical formulae in some calculus such as FOPL or an extended FOPL.

Lexical Semantics Approaches

The semantic grammars described in Section 12.2 are one form of approach based on the use of lexical semantics. With this approach, input sentences are transformed through the use of domain dependent semantic rewrite rules which create the target knowledge structures. A second example of an informal lexical-semantic approach is one which uses conceptual dependency theory. Conceptual dependency structures provide a form of linked knowledge that can be used in larger structures such as scenes and scripts.

The construction of conceptual dependency structures is accomplished without performing any direct syntactic analysis. Making the jump between utterances and

ACTOR:	(a PP with animate attributes)
OBJECT:	(a PP)
ACTION:	(one of the primitive acts with tense)
DIRECTION:	(from-to direction of the action)
INSTRUMENT:	(object with which the act is performed)
LOCATION:	(event location information)
TIME:	(time of the event information)

Figure 12.13 Conceptual dependency structure.

these structures requires that more information be contained in the lexicon. The lexicon entries must include word sense and other information which relate the words to a number of primitive semantic categories as well as some syntactic information.

Recall from Chapter 7 that conceptualizations are either events or object states. Event structures include objects and their attributes, picture producers (PPs) or actors, actions, direction of action (to or from) and sometimes instruments that participate in the actions, and the location and time of the event. These items are collected together in a slot-filler structure as depicted in Figure 12.13.

Verbs in the input string are a dominant factor in building conceptual dependency structures because they denote the event action or state. Consequently, lexicon entries for verbs will be more extensive than other entry types. They will contain all possible senses, tense, and other information. Each verb maps to one of the primitive actions: ATRANS, ATTEND, CONC, EXPEL, GRASP, INGEST, MBUILD, MOVE, MTRANS, PROPEL, PTRANS, and SPEAK. Each primitive action will also have an associated tense: past, present, future, conditional, continuous, interrogative, end, negation, start, and timeless.

The basic process followed in building conceptual dependency structures is simply the three steps listed below.

1. Obtain the next lexical item (a word or phrase).
2. Access the lexical entry for the item and obtain the associated tests and actions.
3. Perform the specified actions given with the entry.

Three types of tests are performed in Step 2.

1. If a certain lexical entry is found, the indicated action is performed. This corresponds to true (non-nil in LISP).
2. Specific word orderings are checked as the structure is being built and actions initiated as a result of the orderings. For example, if a PP follows the last word parsed, action is initiated to fill the object slot.
3. Checks are made for specific words or phrases and, if found, specified actions taken. For example, if an intransitive verb such as listen is found, an action

would be initiated to look for associated words which complete the phrase beginning with to or for.

For the above tests, there are four types of actions taken.

1. Adding additional structure to a partially built conceptual dependency.
2. Filling a slot with a substructure.
3. Activating another action.
4. Deactivating an action.

These actions build up the conceptual dependency structure as the input string is parsed. For example, the action taken for a verb like drank would be to build a substructure for the primitive action INGEST with unfilled slots for ACTOR, OBJECT, and TENSE.

```
(INGEST (ACTOR nil)
        (OBJECT nil)
        (TENSE past))
```

Subsequent words in the input string would initiate actions to add to this structure and fill in the empty ACTOR and OBJECT slots. Thus, a simple sentence like

The boy drank a soda

would be transformed through a series of test and action steps to produce a structure such as the following.

```
(INGEST (ACTOR (PP (NAME boy) (CLASS PHYS-OBJ)
                   (TYPE ANIMATE) (REF DEF)))
        (OBJECT (PP (NAME soda) (CLASS PHYS-OBJ)
                   (TYPE INANIMATE) (REF INDEF)))
        (TENSE PAST))
```

Compositional Semantics Approaches

In the compositional semantics approach, the meaning of an expression is derived from the meanings of the parts of the expression. The target knowledge structures constructed in this approach are typically logic expressions such as the formulas of FOPL. The LUNAR system developed by Woods (1978) uses this approach. The input strings are first parsed using an ATN from which a syntactic tree is output. This becomes the input to a semantic interpreter which interprets the meaning of the syntactic tree and creates the semantic representations.

As an example, suppose the following declaration is submitted to LUNAR.

Sample24 contains silicon

This would be parsed, and the following tree structure would be output from the ATN:

```
(S DCL
   (NP   (N (Sample24)))
   (AUX (TENSE (PRESENT)))
   (VP   (V (contain))
         (NP (N (silicon)))))
```

Using this structure, the semantic interpreter would produce the predicate clause

```
(CONTAIN sample24 silicon)
```

which has the FOPL meaning you would expect.

The interpreter used in LUNAR is driven by semantic pattern → action interpretation rules. The rule that builds the CONTAIN predicate is selected whenever the input tree has a verb of the form have or contain and a sample as the subject and either chemical element, isotope, or oxide as an object. The action of the rule states that such a sentence is to be interpreted as an instance of the schema (CONTAIN *x y*) with *x* and *y* being replaced by the ATN's interpretation of subject noun phrase and object respectively.

LUNAR is also capable of performing quantification of variables in expressions. The quantifiers are an elaboration of those used in FOPL. They include the standard existential and universal quantifiers "for every" and "for some," as well as others such as "for the," "exactly," "the single," "more than," "at least," and so on. For example, an expression with universal quantification would appear as

```
(For Every X (SEQ samples):CONTAIN X Overall silicon)
```

12.6 NATURAL LANGUAGE GENERATION

It is sometimes claimed that language generation is the exact inverse of language understanding. While it is true the two processes have many differences, it is an over simplification to claim that they are exact opposites.

The generation of natural language is more difficult than understanding it, since a system must not only decide what to say, but how the utterances should be stated. A generation system must decide which form is better (active or passive), which words and structures best express the intent, and when to say what. To produce expressions that are natural and close to humans requires more than rules of syntax, semantics, and discourse. In general, it requires that a coherent plan be developed to carry out multiple goals. A great deal of sophistication goes into the simplest types of utterances when they are intended to convey different shades of meanings and emotions. A participant in a dialog must reason about a hearer's understanding and his or her knowledge and goals. During the dialog, the system must maintain proper focus and formulate expressions that either query, explain, direct, lead or just follow the conversation as appropriate.

The study of language generation falls naturally into three areas: (1) the determination of content, (2) formulating and developing a text utterance plan, and (3) achieving a realization of the desired utterances.

Content determination is concerned with what details to include in an explanation, a request, a question or argument in order to convey the meanings set forth by the goals of the speaker. This means the speaker must know what the hearer already knows, what the hearer needs to know, and what the hearer wants to know. These topics are related to the domain, task, and discourse context described above. *Text planning* is the process of organizing the content to be communicated so as to best achieve the goals of the speaker. *Realization* is the process of mapping the organized content to actual text. This requires that specific words and phrases be chosen and formulated into a syntactic structure.

Until about 1980, not much work had been done beyond single sentence generation. Understanding and generation was performed with a single piece of isolated text without much regard given to context and consideration of the hearer. Following this early work, a few comprehensive systems were developed. To complete this section, we describe the basic ideas behind two of these systems. They take different approaches to those taken by the lexical and compositional semantics understanding described in the previous section.

Language Planning and Generation with KAMP

KAMP is a knowledge and modalities planner developed for the generation of natural language text. Developed by Douglas Appelt (1985), KAMP simulates the behavior of an expert robot named Rob (a terminal) assisting John (a person) in the disassembly and repair of air compressors.

KAMP uses a planner and a data base of knowledge in (modal) logical form. The knowledge includes domain knowledge, world knowledge, linguistic knowledge, and knowledge about the hearer. A description of actions and action summaries are available to the planner. Given a goal, the planner uses heuristics to build and refine a plan in the form of a procedural network. Other procedures act as critics of the plans and help to refine them. If a plan is completed, a deduction system is used to prove that the sequence of actions do, in fact, achieve the goal. If the plan fails, the planner must do further searching for a sequence of actions that will work. A completed plan states the knowledge and intentions of the agent, the robot

Rob. This is the first step in producing the output text. The process can be summarized as follows.

Suppose KAMP has determined the immediate goal to be the removal of the compressor pump from the platform.

```
True(¨Attached(pump platform))
```

KAMP first formulates and refines a plan that John adopt Rob's plan to remove the pump from the platform. The first part of Rob's plan suggests a request for John to remove the pump leading to the expression

```
INTENDS(john, REMOVE(pump platform))
```

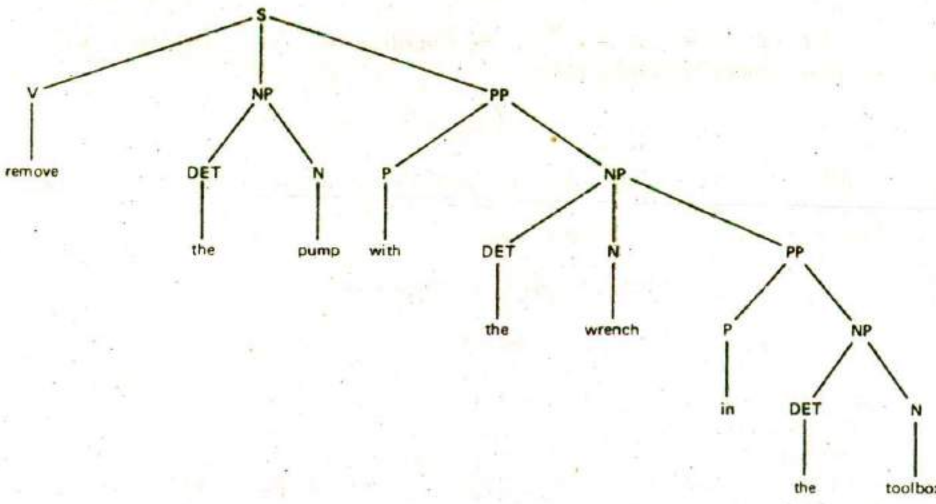
After axioms are used to prove that actions in the initial summary plan are successful, the request is expanded to include details for the pump removal. Rob decides that John will know he is near the platform and that he knows where the toolbox is located, but that he does not know what tool to use. Rob, therefore, determines that John will not need to be told about the platform, but that he must be informed, with an imperative statement, to remove the pump with a wrench in the toolbox.

```
INTENDS(john, REMOVE(pump platform))
LOCATION(john) = LOCATION(platform)

DO(john, REMOVE(pump platform))

DO(rob, REQUEST(john, REMOVE(pump platform)))
DO(rob, COMMAND(john, REMOVE(pump platform)))

DO(rob, INFORM(john (TOOL(wrench))))
DO(rob, INFORM(john LOCATION (wrench) =
                            LOCATION(tool-box)))
```

The next step is for Rob to plán speech acts to realize the request. This requires linguistic knowledge of the structure to use for an imperative request, in this case, that the sentence should have the form V NP (PP)* (recall that * stands for optional repetition). Words to complete the output string are then selected and ordered accordingly.

```
DO(rob REQUEST(john, REMOVE(pump platform)))
DO(rob COMMAND(john, REMOVE(pump platform)))
DO(rob INFORM(john, TOOL(wrench)))
DO(rob INFORM(john, LOCATION(wrench) =
                            LOCATION(tool-box)))
```

This leads to the generation of a sentence with the following tree structure.



The overall process of planning and formulating the final sentence "Remove the pump with the wrench in the toolbox" is very involved and detailed. It requires planning and plan verification for content, selecting the proper structures, selecting senses, mood, tense, the actual words, and a final ordering. All of the steps must be constrained toward the realization of the (possibly multiple) goals set forth. It is truly amazing we accomplish such acts with so little effort.

Generation from Conceptual Dependency Structures

Niel Goldman (Schank et al., 1973) developed a generation component called BABEL which was used as part of several language understanding systems built by Schank and his students (SAM, MARGIE, QUALM, and so on). This component worked in conjunction with an inference component to determine responses to questions about short news and other stories.

Given the general content or primitive event for the response, BABEL selects and builds an appropriate conceptual dependency structure which includes the intended word senses. A modified ATN is then used to generate the actual word string for output.

To determine the proper word sense, BABEL uses a discrimination net. For example, suppose the system is told a story about Joe going into a fast-food restaurant, ordering a sandwich and a soft drink in a can, paying, eating, and then leaving. After the understanding part of the system builds the conceptual dependency and script structures for the story, questions about the events could be posed. If asked what Joe had in the restaurant, BABEL would first need to determine the conceptual

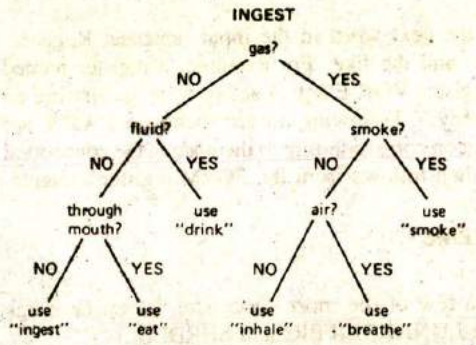
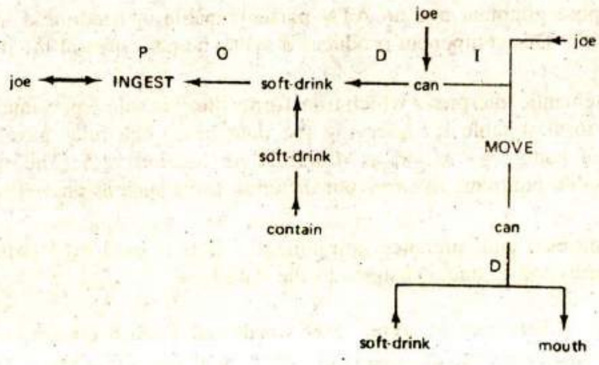


Figure 12.14 Discrimination net for INGEST.

category of the question in order to select the proper conceptual dependency pattern to build. The verb in the query determines the appropriate primitive categories of eat and drink as being INGEST. To determine the correct sense of INGEST as eat and drink a discrimination net like that depicted in Figure 12.14 would be used. A traversal of the discrimination net leads to eat and drink, using the relation from have and sandwich as being taken through the mouth and soft drink as fluid.

Once a conceptual dependency framework has been selected, the appropriate words must be chosen and the slots filled. Functions are used to operate on the net to complete it syntactically to obtain the correct tense, mood, form, and voice. When completed, a modified ATN is then used to transform the conceptual dependency structure into a surface sentence structure for output.

The final conceptual dependency structure passed to the ATN would appear as follows.



An ATN used for text generation differs from one used for analysis. In particular, the registers and arcs must be different. The value of the register contents (denoted as @ in the previous section) corresponds to a node or arc in the conceptual dependency (or other type) network rather than the next word in the input sentence. Registers will be present to hold tense, voice, and the like. For example, a register named FORM might be set to past and a register VOICE set to active when generating an active sentence like "Joe bought candy." Following an arc such as a CAT/V arc means there must be a word in the lexicon corresponding to the node in the conceptual dependency. The tense of the word then follows from the FORM register contents.

12.7 NATURAL LANGUAGE SYSTEMS

In this section, we briefly describe a few of the more successful natural language understanding systems. They include LUNAR, LIFER, and SHRDLU.

The LUNAR System

The LUNAR system was designed as a language interface to give geologists direct access to a data base containing information on lunar rock and soil compositions obtained during the NASA Apollo-11 moon landing mission. The design objective was to build a system that could respond to natural queries received from geologists such as

What is the average concentration of aluminum in high-alkali rocks?
What is the average of the basalt?
In which samples has apatite been identified?

LUNAR has three main components:

1. A general purpose grammar and an ATN parser capable of handling a large subset of English. This component produces a syntactic parse tree of the input sentence.
2. A rule-driven semantic interpreter which transforms the syntactic representation into a logical form suitable for querying the data base. The rules have the general form of pattern → action as described in Section 12.5. The rules produce disposable programs to carry out different tasks such as answering a query.
3. A data base retrieval and inference component which is used to determine answers to queries and to make changes to the data base.

The system has a dictionary of some 3500 words, an English grammar and two data bases. One data base contains a table of chemical analyses of about 13,000 entries, and the other contains 10,000 indexed document topics. LUNAR uses a meaning representation language which is an extended form of FOPL. The language uses (1) designators which name objects or classes of objects like nouns, variables, and classes with range quantifiers, (2) propositions that can be true or false, that are connected with logical operators and, or, not, and quantification identifiers, and (3) commands which carry out specific actions (like TEST which tests the truth value of propositions against given arguments (TEST (CONTAIN sample24 silicon))).

Although never fully implemented, the LUNAR project was considered an operational success since it related to a real world problem in need of a solution. It failed to parse or find the correct semantic interpretation on only about 10% of the questions presented to it.

The LIFER System

LIFER (Language Interface Facility with Ellipsis and Recursion) was described briefly in Section 12.2 under semantic grammars. It was developed by Gary Hendrix (1978) and his associates to be used as a development aid and run-time language interface to other systems such as a data base management system. Among its special features are spelling corrections, processing of elliptical inputs, and the ability of the run-time user to extend the language through the use of paraphrase.

LIFER consists of two major components, a set of interactive functions for language specifications and a parser. The specification functions are used to define an application language as a subset of English that is capable of interacting with existing software. Given the language specification, the parser interprets the language inputs and translates them into appropriate structures that interact with the application software.

In using a semantic grammar, LIFER systems incorporate much semantic information within the syntax. Rather than using categories like NP, VP, N, and V, LIFER uses semantic categories like <SHIP-NAME> and <ATTRIBUTE> which match ship names or attributes. In place of syntactic patterns like NP VP, semantic patterns like What is the <ATTRIBUTE> of <SHIP>? are used. For each such pattern, the language definer supplies an expression with which to compute the interpretations of instances of the pattern. For example, if LIFER were used as the front end for a database query system, the interpretation would be for a database retrieval command.

LIFER has proven to be effective as a front end for a number of systems. The main disadvantage, as noted earlier, is the potentially large number of patterns that may be required for a system which requires many, diverse patterns.

The SHRDLU System

SHRDLU was developed by Terry Winograd as part of his doctoral work at M.I.T. (1972, 1986). The system simulates a simple robot arm that manipulates blocks on a table. During a dialog which is interactive, the system can be asked to manipulate the block objects and build stacks or put things into a box. It can be questioned about the configuration of things on the table, about events that have transpired during the dialog, and even about its reasoning. It can also be told facts which are added to its knowledge base for later reasoning.

The unique aspect of the system is that the meanings of words and phrases are encoded into procedures that are activated by input sentences. Furthermore, the syntactic and semantic analysis, as well as the reasoning process are more closely integrated.

The system can be roughly divided into four component domains: (1) a syntactic parser which is governed by a large English (systemic type) grammar, (2) a semantic component of programs that interpret the meanings of words and structures, (3) a cognitive deduction component used to examine consequences of facts, carry out commands, and find answers, and (4) an English response generation component. In addition, there is a knowledge base containing blocks world knowledge, and a model of its own reasoning process, used to explain its actions.

Knowledge is represented with FOPL-like statements which give the state of the world at any particular time and procedures for changing and reasoning about the state. For example, the expressions

```
(IS b1 block)
(IS b2 pyramid)
(AT b1 (LOCATION 120 120 0))
(SUPPORT b1 b2)
(CLEARTOP b2)
(MANIPULATE b1)
(IS blue color)
```

contain facts describing that b1 is a block, b2 is a pyramid, and b1 supports b2. There are also procedural expressions to perform different tasks such as clear the top or manipulate an object. The CLEARTOP expression is essentially a procedure that first checks to see if the object X supports an object Y. If so, it goes to GET-RID-OF Y and checks again. Integrating the parts of the understanding process with procedural knowledge has resulted in an efficient and effective understanding system. Of course, the domain of SHRDLU is very limited and closed, greatly simplifying the problem.

12.8 SUMMARY

Understanding and generating human language is a difficult problem. It requires a knowledge of grammar and language, of syntax and semantics, of what people know and believe, their goals, the contextual setting, pragmatics, and world knowledge.

We began this chapter with an overview of topics in linguistics, including sentence types, word functions, and the parts of speech. The different forms of knowledge used in natural language understanding were then presented: phonological, morphological, syntactic, semantic, pragmatic, and world. Three general approaches have been followed in developing natural language systems: keyword and pattern matching, syntactic and semantic directed analysis, and matching real world scenarios. Most of the material in this chapter followed the syntactic and semantic directed approach.

Grammars were formally introduced, and the Chomsky hierarchy was presented. This was followed with a description of structural representations for sentences, the phrase marker. Four additional extended grammars were briefly described. One was the transformational grammars, an extension of generative grammars. Transformational grammars include tree manipulation rules that permit the construction of deeper semantic structures than the generative grammars. Case, semantic, and systemic grammars were given as examples of grammars that are also more semantic oriented than the generative grammars.

Lexicons were described, and the role they play in NL systems given. Basic parsing techniques were examined. We locked at simple transition networks, recursive transition networks, and the versatile ATN. The ATN includes tests and actions as part of the arc components and special registers to help in building syntactic structures. With an ATN, extensive semantic analysis is even possible. We defined top-down, bottom-up, deterministic, and nondeterministic parsing methods, and an example of a simple PROLOG parser was also discussed.

We next looked at the semantic interpretation process and discussed two broad approaches, namely the lexical and compositional semantic approaches. These approaches are also identified with the type of target knowledge structures generated. In the compositional semantics approach, logical forms were generated, whereas in the lexical semantics approach, conceptual dependency or similar network structures are created.

Language generation is approximately the opposite of the understanding analysis process, although more difficult. Not only must a system decide what to say but how to say it. Generation falls naturally into three areas, content determination, text planning, and text realization. Two general approaches were presented. They are like the inverses of the lexical and compositional semantic analysis processes. The KAMP system uses an elaborate planning process to determine what, when, and how to state some concepts. The system simulates a robot giving advice to a human helper in the repair of air compressors. At the other extreme, the BABEL system generates output text from conceptual dependency and script structures.

We concluded the chapter with a look at three systems of somewhat disparate architectures: the LUNAR, LIFER, and SHRDLU systems. These systems typify the state-of-the-art in natural language processing systems.

EXERCISES

12.1. Derive a parse tree for the sentence "Bill loves the frog," where the following rewrite rules are used.

```
S   → NP VP
NP  → N
NP  → DET N
VP  → V NP
```

```
DET → the
V   → loves
N   → bill | frog
```

12.2. Develop a parse tree for the sentence "Jack slept on the table" using the following rules.

```
S    → NP VP
NP   → N
NP   → DET N
VP   → V PP
PP   → PREP NP
N    → jack | table
V    → slept
DET  → the
PREP → on
```

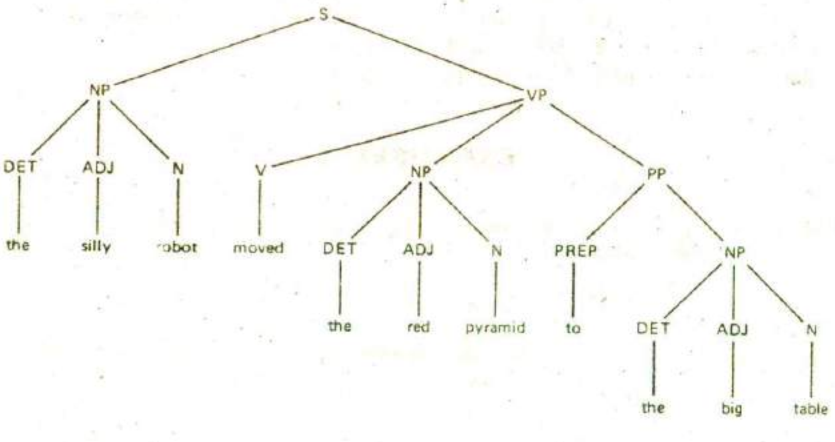
12.3. Give an example of each of the four types 0, 1, 2, and 3 for Chomsky's hierarchy of grammers.

12.4. Modify the grammer of Problem 12.1 to allow the NP (noun phrase) to have zero to many adjectives.

12.5. Explain the main differences between the following three grammars and describe the principal features that could be used to develop specifications for a syntactical recognition program. Consult additional references for more details regarding each grammar.
Chomsky's Transformational Grammar
Fillmore's Case Grammar
Systemic Grammars

12.6. Draw an ATN to implement the grammer of Problem 12.1.

12.7. Given the following parse tree, write down the corresponding context free grammer.



12.8. Create a LISP data structure to model a simple lexicon similar to the one depicted in Figure 12.6.

12.9. Write a LISP match program which checks an input sentence for matching words in the lexicon of the previous problem.

12.10. Derive an ATN for the parse tree of Problem 12.7.

12.11. Derive an ATN (graph) to implement the parse tree of Problem 12.1.

12.12. Determine if the following sentences will be accepted by the grammar of Problem 12.6.
(a) The green green grass of the home
(b) The red car drove in the fast lane.

12.13. Write PROLOG rules to implement the grammar used to derive the parse tree of Problem 12.7. Omit rules for the individual word categories (like noun ([ball | X],X)). Generate a syntax tree using one output parameter.

12.14. Write a PROLOG program that will take grammar rules in the following format:

$$NT \rightarrow (NT \mid T)^*$$

where NT is any nonterminal, T is any terminal, and Kleene star (*) signifies any number of repetitions, and generate the corresponding top-down parser; that is,

```
sentence → noun_phrase, verb_phrase
determiner → [the]
```

will generate the following:

```
sentence(I,O) :- noun_phrase(I,R), verb_phrase(R,O).
determiner([the|X],X) :-!.
```

12.15. Modify the program in Problem 12.12 to accept extra arguments used to return meaningful knowledge structures.

```
sentence(sentence(NP,VP)) → noun_phrase(NP), verb_phrase (VP).
```

12.16. Write a LISP program which uses property lists to create the recursive transition network depicted in Figure 12.9. Each node should be given a name such as S1, N1, and P1 and associated with a list of arc and node pairs emanating from the node.

12.17. Write a recursive program in LISP which tests input sentences for the RTN developed in the previous problem. The program should return *t* if the sentence is acceptable, and nil if not.

12.18. Modify the program of Problem 12.15 to accept sentences of the type depicted in Figure 12.12.

12.19. Write an ATN type of program as depicted in Figure 12.12 which builds structures like those of Figure 12.13.

12.20. Describe in detail the differences between language understanding and language generation. Explain the problems in developing a program which is capable of carrying on a dialog with a group of people.

12.21. Give the processing steps required and corresponding data structures needed for a robot named Rob to formulate instructions for a helper named John to complete a university course add-drop request form.

12.22. Give the conceptual dependency graph for the sentence "Mary drove her car to school" and describe the steps required for a program to transform the sentence to an internal conceptual dependency structure.

13 Pattern Recognition

One of the most basic and essential characteristics of living things is the ability to recognize and identify objects. Certainly all higher animals depend on this ability for their very survival. Without it they would be unable to function even in a static, unchanging environment.

In this chapter we consider the process of computer pattern recognition, a process whereby computer programs are used to recognize various forms of input stimuli such as visual or acoustic (speech) patterns. This material will help to round out the topic of natural language understanding when speech, rather than text, is the language source. It will also serve as an introduction to the following chapter where we take up the general problem of computer vision.

Although some researchers feel that pattern recognition should no longer be considered a part of AI, we believe many topics from pattern recognition are essential to an understanding and appreciation of important concepts related to natural language understanding, computer vision, and machine learning. Consequently, we have included in this chapter a selected number of those topics believed to be important.

13.1 INTRODUCTION

Recognition is the process of establishing a close match between some new stimulus and previously stored stimulus patterns. This process is being performed continually throughout the lives of all living things. In higher animals this ability is manifested in many forms at both the conscious and unconscious levels, for both abstract as well as physical objects. Through visual sensing and recognition, we identify many special objects, such as home, office, school, restaurants, faces of people, handwriting, and printed words. Through aural sensing and recognition, we identify familiar voices, songs and pieces of music, and bird and other animal sounds. Through touch, we identify physical objects such as pens, cups, automobile controls, and food items. And through our other senses we identify foods, fresh air, toxic substances and much else.

At more abstract levels of cognition, we recognize or identify such things as ideas (electromagnetic radiation phenomena, model of the atom, world peace), concepts (beauty, generosity, complexity), procedures (game playing, making a bank deposit), plans, old arguments, metaphors, and so on.

Our pervasive use of and dependence on our ability to recognize patterns has motivated much research toward the discovery of mechanical or artificial methods comparable to those used by intelligent beings. The results of these efforts to date have been impressive, and numerous applications have resulted. Systems have now been developed to reliably perform character and speech recognition; fingerprint and photograph identifications; electroencephelogram (EEG), electrocardiogram (ECG), oil log-well, and other graphical pattern analyses; various types of medical and system diagnoses; resource identification and evaluation (geological, forestry, hydrological, crop disease); and detection of explosive and hostile threats (submarine, aircraft, missile) to name a few.

Object classification is closely related to recognition. The ability to classify or group objects according to some commonly shared features is a form of class recognition. Classification is essential for decision making, learning, and many other cognitive acts. Like recognition, classification depends on the ability to discover common patterns among objects. This ability, in turn, must be acquired through some learning process. Prominent feature patterns which characterize classes of objects must be discovered, generalized, and stored for subsequent recall and comparison.

We do not know exactly how humans learn to identify or classify objects, however, it appears the following processes take place:

New objects are introduced to a human through activation of sensor stimuli. The sensors, depending on their physical properties, are sensitive in varying degrees to certain attributes which serve to characterize the objects, and the sensor output tends to be proportional to the more prominent attributes. Having perceived a new object, a cognitive model is formed from the stimuli patterns and stored in memory. Recurrent experiences in perceiving the same or similar objects strengthen and refine the similarity patterns. Repeated perception results in the formation of generalized or archetype models of object classes which become useful in matching, and hence recognition, of similar objects.

13.2 THE RECOGNITION AND CLASSIFICATION PROCESS

In artificial or mechanical recognition, essentially the same steps as noted above must be performed. These steps are illustrated in Figure 13.1 and summarized below:

Step 1. Stimuli produced by objects are perceived by sensory devices. The more prominent attributes (such as size, shape, color, and texture) produce the strongest stimuli. The values of these attributes and their relations are used to characterize an object in the form of a pattern vector **X**, as a string generated by some grammar, as a classification tree, a description graph, or some other means of representation. The range of characteristic attribute values is known as the measurement space M.

Step 2. A subset of attributes whose values provide cohesive object grouping or clustering, consistent with some goals associated with the object classifications, are selected. Attributes selected are those which produce high intraclass and low interclass groupings. This subset represents a reduction in the attribute space dimensionality and hence simplifies the classification process. The range of the subset of attribute values is known as the feature space F.

Step 3. Using the selected attribute values, object or class characterization models are learned by forming generalized phototype descriptions, classification rules, or decision functions. These models are stored for subsequent recognition. The range of the decision function values or classification rules is known as the decision space D.

Step 4. Recognition of familiar objects is achieved through application of the rules learned in Step 3 by comparison and matching of object features with the stored models. Refinements and adjustments can be performed continually thereafter to improve the quality and speed of recognition.

There are two basic approaches to the recognition problem, (1) the decision-theoretic approach and (2) the syntactic approach.

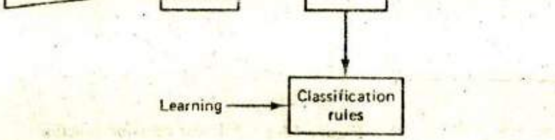


Figure 13.1 The pattern recognition process.

Decision Theoretic Classification

The decision theoretic approach is based on the use of decision functions to classify objects. A decision function maps pattern vectors **X** into decision regions of D. More formally, this problem can be stated as follows.

1. Given a universe of objects $0 = \{o_1, o_2, \ldots, o_n\}$, let each o_i have k observable attributes and relations expressable as a vector $\mathbf{V} = (v_1, v_2, \ldots, v_k)$.
2. Determine (a) a subset of $m \leq k$ of the v_i, say $\mathbf{X} = (x_1, x_2, \ldots, x_m)$ whose values uniquely characterize the o_i, and (b) $c \geq 2$ groupings or classifications of the o_i which exhibit high intraclass and low interclass similarities such that a decision function $d(\mathbf{X})$ can be found which partitions D into c disjoint regions. The regions are used to classify each o_i as belonging to at most one of the c classes.

Determining the feature attributes and decision regions requires stipulating or learning mappings from the measurement space M to the feature space F and then a mapping from F to the classification or decision space D,

$$\mathrm{M} \rightarrow \mathrm{F} \rightarrow \mathrm{D}$$

When there are only two classes, say C_1 and C_2, the values of the object's pattern vectors may tend to cluster into two disjoint groups. In this case, a linear decision function $d(\mathbf{X})$ can often be used to determine an object's class. For example, when the classes are clustered as depicted in Figure 13.2, a linear decision function d is adequate to classify unknown objects as belonging to either C_1 or C_2, where

$$d(\mathbf{X}) = w_1x_1 + w_2x_2 + w_3$$

The constants w_i in d are parameters or weights that are adjusted to find a separating line for the classes. When a function such as d is used, an object is classified as belonging to C_1 if its pattern vector is such that $d(\mathbf{X}) < 0$, and as

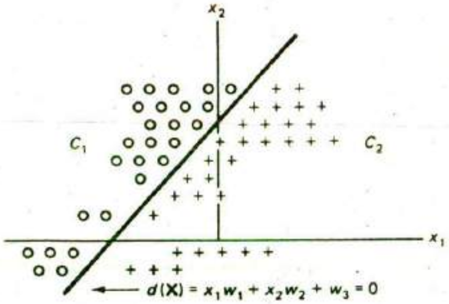


Figure 13.2 A linear decision function.

belonging to class C_2 when $d(\mathbf{X}) > 0$. When $d(\mathbf{X}) = 0$ the classification is indeterminate, so either (or neither) class may be selected.

When class reference vectors, prototypes $\mathbf{R}_j$, $j = 1, \ldots, c$ are available, decision functions can be defined in terms of the distance of the $\mathbf{X}$ from the reference vectors. For example, the distance

$$d_i(\mathbf{X}) = (\mathbf{X} - \mathbf{R}_i)'(\mathbf{X} - \mathbf{R}_i)$$

could be computed for each class C_i, and class C_k would then be chosen when $d_k = \min_i\{C_i\}$.

For the general case of $c \geq 2$ classes, $C_1, C_2, \ldots, C_c$, a decision function may be defined for each class $d_1, d_2, \ldots, d_c$. A class decision rule in this case would be defined to select class C_j when

$$d_j(\mathbf{X}) < d_i(\mathbf{X}) \text{ for } i,j = 1, 2, \ldots, c, \text{ and } i \neq j.$$

When a line d (or more generally a hyperplane in n-space) can be found that separates classes into two or more groups as in the case of Figure 13.2, we say the classes are linearly separable. Classes that overlap each other or surround one another, as in Figure 13.3, cannot generally be classified with the use of simple linear decison functions. For such cases, more general nonlinear (or piecewise-linear) functions may be required. Alternatively, some other selection technique (like heuristics) may be needed.

The decision function approach described above is an example of deterministic recognition since the x_i are deterministic variables. In cases where the attribute values are affected by noise or other random fluctuations, it may be more appropriate to define probabilistic decision functions. In such cases, the attribute vectors $\mathbf{X}$ are treated as random variables, and the decision functions are defined as measures of likelihood of class inclusion. For example, using Bayes' rule, one can compute the conditional probability $P(C_i|\mathbf{X})$ that the class of an object o_j is C_i given the observed value of $\mathbf{X}$ for o_j. This approach requires a knowledge of the prior probability $P(C_i)$, the probability of the occurrence of samples from C_i, as well as $P(\mathbf{X}|C_i)$.

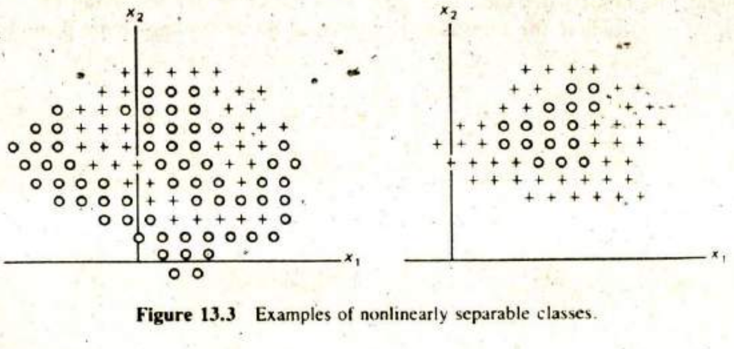


Figure 13.3 Examples of nonlinearly separable classes.

(Note that the C_i are treated like random variables here. This is equivalent to the assumption made in Bayesian classification where the distribution parameter Θ is assumed to be a random variable since C_i may be regarded as a function of Θ). A decision rule for this case is to choose class C_j if

$$P(C_j \mid \mathbf{X}) > P(C_i \mid \mathbf{X}) \text{ for all } i \neq j.$$

A more comprehensive probabilistic approach is one which is based on the use of a loss or risk Bayesian function where the class is chosen on the basis of minimum loss or risk. Let the loss function L_{ij} denote the loss incurred by incorrectly classifying an object actually belonging to class C_i as belonging to C_j. When L_{ij} is a constant for all $i, j, i \neq j$, a decision rule can be formulated using the likelihood ratio defined as (see Chapter 6)

$$\frac{P(\mathbf{X} \mid C_k)}{P(\mathbf{X} \mid C_j)}$$

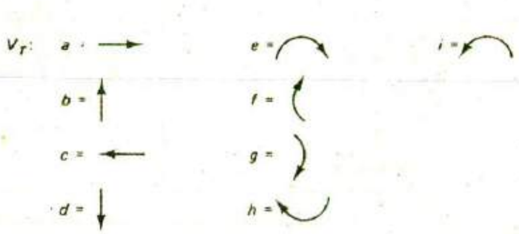
The rule is to choose class C_k whenever the relation

$$\frac{P(\mathbf{X} \mid C_k)}{P(\mathbf{X} \mid C_j)} > \frac{P(C_j)}{P(C_k)} \quad \text{holds for all } j \neq k.$$

Probabilistic decision rules may be constructed as either parametric or nonparametric depending on knowledge of the distribution forms, respectively. For a comprehensive treatment of these methods see (Duda and Hart, 1973) or (Tou and Gonzales, 1974).

Syntactic Classification

The syntactic recognition approach is based on the uniqueness of syntactic "structure" among the object classes. With this approach, a grammar similar to the grammars defined in Chapter 10 or the generative grammars of Chapter 12 is defined for object descriptions. Instead of defining the grammar in terms of an alphabet of characters or terminal words, the vocabulary is based on shape primitives. For example, the objects depicted in Figure 13.4 could be defined using the grammar $G(v_n, v_t, p, s)$, where the terminals v_t consist of the following shape primitives.



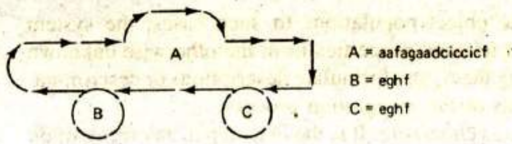


Figure 13.4 Syntactic characterization of objects.

Using syntactic analysis, that is parsing and analyzing the string structures, classification is accomplished by assigning an object to class C_i when the string describing it has been generated by the grammar G_i. This requires that the string be recognized as a member of the language $L(G_i)$. If there are only two classes, it is sufficient to have a single grammar G (two grammars are needed when strings of neither class can occur).

When classification for $c \geq 2$ classes is required, $c - 1$ (or c) different grammars are needed for class recognition. The decision functions in this case are based on grammar recognition functions which choose class C_j if the pattern string is found to be generated by grammar G_j, that is, if it is a member of $L(G_j)$. Patterns not recognized as a member of a defined language are indeterminate.

When patterns are noisy or subject to random fluctuations, ambiguities may occur since patterns belonging to different classes may appear to be the same. In such cases, stochastic or fuzzy grammars may be used. Classification for these cases may be made on the basis of least cost to transform an input string into a valid recognizable string, by the degree of class set inclusion or with a similarity measure using one of the methods described in Chapter 10.

13.3 LEARNING CLASSIFICATION PATTERNS

Before a system can recognize objects, it must possess knowledge of the characteristic features for those objects. This means that the system designer must either build the necessary discriminating rules into the system or the system must learn them. In the case of a linear decision function, the weights that define class boundaries must be predefined or learned. In the case of syntactic recognition, the class grammars must be predefined or learned.

Learning decision functions, grammars, or other rules can be performed in either of two ways, through supervised learning or unsupervised learning. Supervised learning is accomplished by presenting training examples to a learning unit. The examples are labeled beforehand with their correct identities or class. The attribute values and object labels are used by the learning component to inductively extract and determine pattern criteria for each class. This knowledge is used to adjust parameters in decision functions or grammar rewrite rules. Supervised learning concepts are discussed in some detail in Part V. Therefore, we concentrate here on some of the more important notions related to unsupervised learning.

In unsupervised learning, labeled training examples are not available and little is known beforehand regarding the object population. In such cases, the system must be able to perceive and extract relevant properties from the otherwise unknown objects, find common patterns among them, and formulate descriptions or descrimination criteria consistent with the goals of the recognition process.

This form of learning is known as *clustering*. It is the first step in any recognition process where discriminating features of objects are not known in advance.

Learning through Clustering

Clustering is the process of grouping or classifying objects on the basis of a close association or shared characteristics. The objects can be physical or abstract entities, and the characteristics can be attribute values, relations among the objects, and combinations of both. For example, the objects might be streets, freeways, and other pathways connecting two points in a city, and the classifications, the pathways which provide fast or slow traversal between the points. At a more abstract level, the objects might be some concept such as the quality of the items purchased. The classifications in this case might be made on the basis of some subjective criteria, such as poor, average, or good.

Clustering is essentially a discovery learning process in which similarity patterns are found among a group of objects. Discovery of the patterns is usually influenced by the environment or context and motivated by some goal or objective (even if only for economy in cognition). For example, finding short-cuts between two frequently visited points is motivated by a desire to reduce the planning effort and transit time between the points. Likewise, developing a notion of quality is motivated by a desire to save time and money or to improve one's appearance.

Given different objectives, the same set of objects would, in general, be clustered differently. If the objective given above for the streets, freeways, and the like were modified to include safe for bicycle riding, a different object classification would, in general, result.

Finding the most meaningful cluster groupings among a set of unknown objects o_i requires that similarity patterns be discovered in the feature space. Clustering is usually performed with the intent of capturing any gestalt properties of a group of objects and not just the commonality of certain attribute values. This is one of the basic requirements of *conceptual* clustering (Chapter 19) where the objects are grouped together as members of a concept class. Procedures for conceptual clustering are based on more than simple distance measures. They must also take into account the context (environment) of the objects as well as the goals or objectives of the clustering.

The clustering problem gives rise to several subproblems. In particular, before an implementation is possible, the following questions must be addressed.

1. What set of attributes and relations are most relevant, and what weights should be given to each? In what order should attributes be observed or measured? (If the observation process is sequential, ordering may influence the effectiveness of the attributes in discriminating among objects.)

2. What representation formalism should be used to characterize the objects?

3. What representation scheme should be used to describe the cluster groupings or classifications? Usually, some simplification results if the single representation trick can be used (the use of a single representation method for both object and cluster descriptions).

4. What clustering criteria is most consistent with and effective in achieving the objectives relative to the context or domain? This requires consideration of an appropriate distance or similarity measure compatible with the description domains noted in 2, above.

5. What clustering algorithms can best meet the criteria in 2 within acceptable time and space complexity bounds?

By now questions such as these should be familiar. They are by no means trivial, but they must be addressed when designing a system. They depend on many complex factors for which the tools of earlier chapters become essential. These problems have been addressed elsewhere; therefore, we focus our attention here on the clustering process.

The clustering process must be performed with a limited set of observations, and checking all possible object groupings for patterns is not feasible except with a small number of objects. This is due to the combinatorial explosion which results in arranging n objects into an unknown number m of clusters.[1] Consequently, methods which examine only the more promising groupings must be used. Establishing such groupings requires the use of some measure of similarity, association, or degree of fit among a set of objects.

When the attribute values are real valued, cluster groupings can sometimes be found with the use of point-to-point or point-to-set distances, probability measures (like using the covariance matrix between two populations), scatter matrices, the sum of squared error distance between objects, or other means (see Chapter 10). In these cases, an object is clustered in class C_i if its proximity to other members of C_i is within some threshold or limiting value.

Many clustering algorithms have been proposed for different tasks. One of the most popular algorithms developed at the Stanford Research Institute by G. H. Ball and D. J. Hall (Anderberg, 1973) is known as the ISODATA method. This method requires that the number of clusters m be specified, and threshold values t_1, t_2, and t_3 be given or determined for use in splitting, merging, or discarding

[1] The number of ways in which n objects can be arranged into m groups is an exponential quantity.

$$S_m = \left(\frac{1}{m!}\right) \sum_k (-1)^{m-k} \binom{m}{k} k^n.$$

When m is unknown, the number of arrangements increases as the sum of the S_m, that is, as S^m. For example when $n = 25$, the number of arrangements is more than $4 * 10^{18}$.

clusters respectively. During the clustering process, the thresholds are used to determine if a cluster should be split into two clusters, merged with other clusters or discarded (when too small). The algorithm is given with the following steps.

1. Select m samples as seed points for initial cluster centers. This can be done by taking the first m points, selecting random points or by taking the first m points which exceed some mutual minimum separation distance d.
2. Group each sample with its nearest cluster center.
3. After all samples have been grouped, compute new cluster centers for each group. The center can be defined as the centroid (mean value of the attribute vectors) or some similar central measure.
4. If the split threshold t_1 is exceeded for any cluster, split it into two parts and recompute new cluster centers.
5. If the distance between two cluster centers is less than t_2, combine the clusters and recompute new cluster centers.
6. If a cluster has fewer than t_3 members, discard the cluster. It is ignored for the remainder of the process.
7. Repeat steps 3 through 6 until no change occurs among cluster groupings or until some iteration limit has been exceeded.

Measures for determining distances and the center location need not be based on ordered variates. They may be one of the measures described in Chapter 10 (including probabilistic or fuzzy measures) or some measure of similarity between graphs, strings, and even FOPL descriptions. In any case, it is assumed each object o_i is described by a unique point or event in the feature space F.

Up to this point we have ignored the problem of attribute scaling. It is possible that a few large valued variables may completely dominate the other variables in a similarity measure. This could happen, for example, if one variable is measured in units of meters and another variable in millimeters or if the range and scale of variation for two variables are widely different. This problem is closely related to the feature selection problem, that is, in the assignment of weights to feature variables on the basis of their importance or relevance. One simple method for adjusting the scales of such variables is to use a diagonal weight matrix $\mathbf{W}$ to transform the representation vector $\mathbf{X}$ to $\mathbf{X}' = \mathbf{WX}$. Thus, for all of the measures described above, one should assume the representation vectors $\mathbf{X}$ have been appropriately normalized to account for scale variations.

To summarize the above process, a subset of characteristic features which represent the o_i are first selected. The features chosen should be good discriminators in separating objects from different classes, relevant, and measurable (observable) at reasonable cost. Feature variables should be scaled as noted above to prevent any swamping effect when combined due to large valued variables. Next, a suitable metric which measures the degree of association or similarity between objects should be chosen, and an appropriate clustering algorithm selected. Finally, during the clustering process, the feature variables may need to be weighted to reflect the relative importance of the feature in affecting the clustering.

13.4 RECOGNIZING AND UNDERSTANDING SPEECH

Developing systems that understand speech has been a continuing goal of AI researchers. Speech is one of our most expedient and natural forms of communication, and so understandably, it is a capability we would like AI systems to possess. The ability to communicate directly with programs offers several advantages. It eliminates the need for keyboard entries and speeds up the interchange of information between user and system. With speech as the communication medium, users are also free to perform other tasks concurrently with the computer interchange. And finally, more untrained personnel would be able to use computers in a variety of applications.

The recognition of continuous waveform patterns such as speech begins with sampling and digitizing the waveforms. In this case the feature values are the sampled points $x_i = f(t_i)$ as illustrated in Figure 13.5.

It is known from information theory that a sampling rate of twice the highest speech frequency is needed to capture the information content of the speech waveforms. Thus, sampling requirements will normally be equivalent to 20K to 30K bytes per second. While this rate of information in itself is not too difficult to handle, this, added to the subsequent processing, does place some heavy requirements on real time understanding of speech.

Following sample digitization, the signals are processed at different levels of abstraction. The lowest level deals with phones (the smallest unit of sound), allophones (variations of the phoneme as they actually occur in words), and syllables. Higher level processing deals with words, phrases, and sentences.

The processing approach may be from the bottom, top, or a combination of both. When bottom processing is used the input signal is segmented into basic speech units and a search is made to match prestored patterns against these units. Knowledge about the phonetic composition of words is stored in a lexicon for comparisons. For the top approach, syntax, semantics (the domain), and pragmatics (context) are used to anticipate which words the speaker is likely to have said and

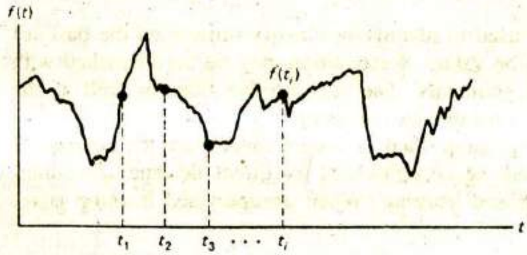


Figure 13.5 Sampling a continuous waveform.

direct the search for recognizable patterns. A combined approach which uses both methods has also been applied successfully.

Early research in speech recognition concentrated on the recognition of isolated words. Patterns of individual words were prestored and then compared to the digitized input patterns. These early systems met with limited success. They were unable to tolerate variations in speaker voices and were highly susceptible to noise. Although important, this early work helped little with the general problem of continuous speech understanding since words appearing as part of a continuous stream differ significantly from isolated words. In continuous speech, words are run together, modified, and truncated to produce a great variation of sounds. Thus, speech analysis must be able to detect different sounds as being part of the same word, but in different contexts. Because of the noise and variability, recognition is best accomplished with some type of fuzzy comparison.

In 1971 the Defense Advanced Research Projects Agency (DARPA) funded a five year program for continuous speech understanding research (SUR). The objective of this research was to design and implement systems that were capable of accepting continuous speech from several cooperative speakers using a limited vocabulary of some 1000 words. The systems were expected to run at slower than real time speeds. A product of this research were several systems including HEARSAY I and II, HARPY, and HWIM. While the systems were only moderately successful in achieving their goals, the research produced other important byproducts as well, particularly in systems architectures, and in the knowledge gained regarding control.

The HEARSAY system was important for its introduction of the blackboard architecture (Chapter 15). This architecture is based on the cooperative efforts of several specialist knowledge components communicating by way of a blackboard in the solution of a class of problems. The specialists are each expert in a different area. For example, speech analysis experts might each deal with a different level of the speech problem. The solution process is opportunistic, with each expert making a contribution when it can. The solution to a given problem is developed as a data structure on the blackboard. As the solution is developed, this data structure is modified by the contributing expert. A description of the systems developed under SUR is given in Barr and Feigenbaum (1981).

13.5 SUMMARY

Pattern recognition systems are used to identify or classify objects on the basis of their attribute and attribute-relation values. Recognition may be accomplished with decision functions or structural grammars. The decision functions as well as the grammars may be deterministic, probabilistic, or fuzzy.

Before recognition can be accomplished, a system must learn the criteria for object recognition. Learning may be accomplished by direct designer encoding, supervised learning, or unsupervised learning. When unsupervised learning is required, some form of clustering may be performed to learn the object class characteristics.

Speech understanding first requires recognition of basic speech patterns. These patterns are matched against lexicon patterns for recognition. Basic speech units such as phonemes are the building blocks for longer units such as syllables and words.

EXERCISES

13.1. Choose three common objects and determine five of their most discriminating visual attributes.

13.2. For the previous problem, determine three additional nonvisual attributes for the objects which are most discriminating.

13.3. Find a linear decision function which separates the following $x-y$ points into two distinct classes.

−1.8	−5,−1	−3.3	−3.0	1.3	−1.1
0,1	3,4	0,0	2,3	−4,−1	−2,3

13.4. Describe how you would design a pattern recognition program which must validate hand written signatures. Identify some potential problem areas.

13.5. Compare the deterministic decision function approach to the probabilistic decision function apporoach in pattern recognition applications. Give examples when each would be the appropriate function to use.

13.6. Define a set of rewrite rules for a grammar for syntactic generation (recognition) of objects such as the object of Figure 13.4.

13.7. Give two examples of unsupervised learning in humans in which they learn to recognize objects through clustering. Describe how different goals can influence the learning process.

13.8. Apply the ISODATA algorithm to find three different clusters among the following $x-y$ data points.

−2.1	−1.3	−1,2	−1,0	0.2	1,7	1.2	1.1
1,0	2.9	2,8	2.5	2,0	3,9	3.7	3.6
4,8	4.6	4.3	4.2	5.4	5.3	5.2	6.3
6,3	7.7	7,5	7.4	7.2	7,0	−1.6	−2,6

13.9. Consider a cluster algorithm which builds clusters of objects by forming small regions in normalized attribute space (spheres in n-dimensional space) about each object, and includes them in a cluster if and only if the sphere overlaps with at least one other neighboring object sphere. Show how such a scheme could be used to partition the attribute space into subspace with nonlinear boundaries.

13.10. Define an alphabet of shape primitives for a syntactic recognition grammar which can be used to recognize the integer characters 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Check to see that the resultant character strings for each character are unique.

13.11. Give two examples where the single representation trick simplifies clustering among unknown objects.

13.12. Compute the number of ways five objects can be arranged into 1, 2, 3, and 4 groups. From this, try to develop an inductive proof of arranging *n* objects into *m* groups.

13.13. Read "High Level Knowledge Sources in Usable Speech Recognition Systems" by Sheryl Young, Alexander Hauptmann, Wayne Ward, Edward Smith, and Philip Werner, in *Communications of the ACM*, Vol. 32, Number 2, Feb., 1989. Summarize some of the more complicated problems associated with general speech recognition.

14 Visual Image Understanding

Vision is perhaps the most remarkable of all of our intelligent sensing capabilities. Through our visual system, we are able to acquire information about our environment without direct contact. Vision permits us to acquire information at a phenomenal rate and at resolutions that are most impressive. For example, one only needs to compare the resolution of a TV camera system to that of a human to see the difference. Roughly speaking, a TV camera has a resolution on the order of 500 parts per square cm, while the human eye has a limiting resolution on the order of some 25×10^6 parts per square cm. Thus, humans have a visual resolution several orders of magnitude better (more than 10,000 times finer) than that of a TV camera. What is even more remarkable is the ease with which we humans sense and perceive a variety of visual images. It is so effortless, we are seldom conscious of the act.

In this chapter, we examine the processes and the problems involved in building computer vision systems. We look at some of the approaches taken thus far and at some of the more successful vision systems constructed to date.

14.1 INTRODUCTION

Because of its wide ranging potential, computer vision has become one of the most intensely studied areas of AI and engineering during the past few decades. Some typical areas of application include the following.

MANUFACTURING

Parts inspection for quality control
Assembly, sorting, dispensing, locating, and packaging of parts

MEDICAL

Screening x-ray, tomographic, ultrasound, and other medical images

DEFENSE

Photo reconnaisance, analysis, and scene interpretation
Target detection, identification, and tracking
Microbe detection and identification
Weapons guidance
Remote and local site monitoring

BUSINESS

Visual document readers
Design tools for engineers and architects
Inspection of labels for identification
Inspection of products for contents and packaging

ROBOTICS

Guidance of welders and spray paint nozzles
Sorting, picking, and bin packing of items
Autonomous guidance of land, air and sea vehicles

SPACE EXPLORATION

Discovery and interpretation of astronomical images
Terrestial image mapping and interpretation for plant disease, mineral deposits, insect infestations, and soil erosion

Vision in an organic system is the process of sensing a pattern of light energy, and developing an interpretation of those patterns. The sensing part of the process consists of selectively gathering light from some area of the environment, focusing and projecting it onto a light sensitive surface, and converting the light into electrochemical patterns of impulses. The perception part of the process involves the transformation and comparison of the transmitted impulse patterns to other prestored patterns

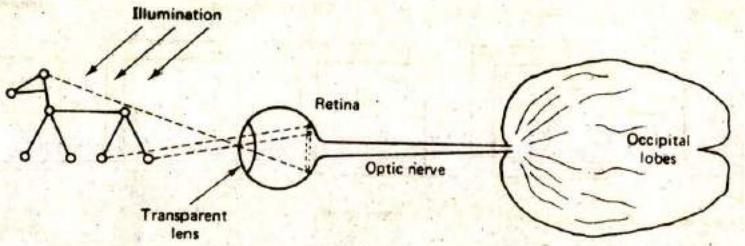


Figure 14.1 The human process of visual interpretation.

together with some form of inference. The basic vision process as it occurs in humans is depicted in Figure 14.1.

Light from illuminated objects is collected by the transparent lens of the eye, focused, and projected onto the retina where some 250 million light sensitive sensors (cones and rods) are excited. When excited, the sensors send impulses through the optic nerve to the visual cortex of the occipital lobes of the brain where the images are interpreted and recognized.

Computer vision systems share some similarities with human visual systems, at least as we now understand them. They also have a number of important differences. Although artificial vision systems vary widely with the specific application, we adopt a general approach here, one in which the ultimate objective is to determine a high-level description of a three-dimensional scene with a competency level comparable to that of human vision systems. Before proceeding farther we should distinguish between a scene and an image of a scene. A scene is the set of physical objects in a picture area, whereas an image is the projection of the scene onto a two-dimensional plane.

With the above objectives in mind, a typical computer vision system should be able to perform the following operations:

1. Image formation, sensing, and digitization
2. Local processing and image segmentation
3. Shape formation and interpretation
4. Semantic analysis and description.

The sequence of these operations is depicted in Figure 14.2.

As we proceed through the processing stages of computer vision, the reader will no doubt be impressed by the similarities and parallels one can draw between vision processing and natural language processing. The image-sensor stage in vision corresponds to speech recognition in language understanding, the low and intermediate processing levels of vision correspond to syntactic and semantic language processing respectively, and high level processing, in both cases, corresponds to the process of building and interpreting high level knowledge structures.

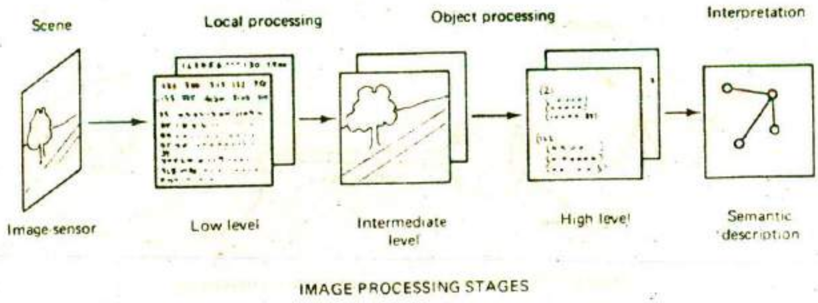


Figure 14.2 Processing stages in computer vision systems.

Vision Processing Overview

The input to a vision system is a two dimensional image collected on some form of light sensitive surface. This surface is scanned by some means to produce a continuous voltage output that is proportional to the light intensity of the image on the surface. The output voltage $f(x, y)$ is sampled at a discrete number of x and y points or pixel (picture element) positions and converted to numbers. The numbers correspond to the gray level intensity for black and white images. For color images, the intensity value is comprised of three separate arrays of numbers, one for the intensity value of each of the basic colors (red, green, and blue).

Thus, through the digitization process, the image is transformed from a continuous light source into an array of numbers which correspond to the local image intensities at the corresponding x-y pixel positions on the light sensitive surface.

Using the array of numbers, certain low level operations are performed, such as smoothing of neighboring points to reduce noise, finding outlines of objects or edge elements, thresholding (recording maximum and minimum values only, depending on some fixed intensity threshold level), and determining texture, color, and other object features. These initial processing steps are ones which are used to locate and accentuate object boundaries and other structure within the image.

The next stage of processing, the intermediate level, involves connecting, filling in, and combining boundaries, determining regions, and assigning descriptive labels to objects that have been accentuated in the first stage. This stage builds higher level structures from the lower level elements of the first stage. When complete, it passes on labeled surfaces such as geometrical objects that may be capable of identification.

High-level image processing consists of identifying the important objects in the image and their relationships for subsequent description as well-defined knowledge structures and hence, for use by a reasoning component.

Special types of vision systems may also require three dimensional processing and analysis as well as motion detection and analysis.

The Objectives of Computer Vision Systems

The ultimate goals of computer image understanding is to build systems that equal or exceed the capabilities of human vision systems. Ideally, a computer vision system would be capable of interpreting and describing any complex scene in complete detail. This means that the system must not only be able to identify a myriad of complex objects, but must also be able to reason about the objects, to describe their function and purpose, what has taken place in the scene, why any visible or implied events occurred, what is likely to happen, and what the objects in the scene are capable of doing.

Figure 14.3 presents an example of a complex scene that humans can interpret well with little effort. It is the objective of many researchers in computer vision to build systems capable of interpreting, describing, and reasoning about scenes of this type in real time. Unfortunately, we are far from achieving this level of competency. To be sure, some interesting vision systems have been developed, but they are quite crude compared to the elegant vision systems of humans.

Like natural language understanding, computer vision interpretation is a difficult problem. The amount of processing and storage required to interpret and describe a complex scene can be enormous. For example, a single image for a high resolution aerial photograph may result in some four to nine million pixels (bytes) of information and require on the average some 10 to 20 computations per pixel. Thus, when several frames must be stored during processing, as many as 100 megabytes of storage may be needed, and more than 100 million computations performed.

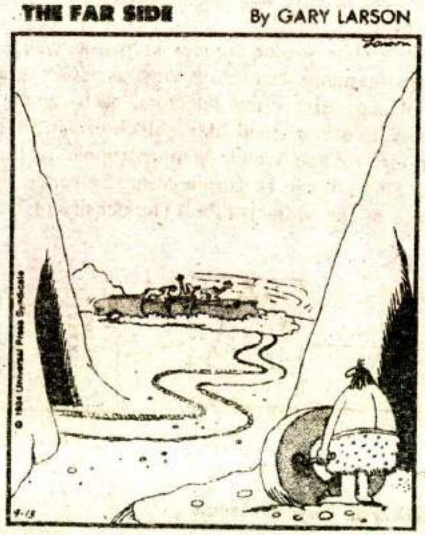


Figure 14.3 Example of a complex scene. (THE FAR SIDE COPYRIGHT 1984 UNIVERSAL PRESS SYNDICATE. Reprinted by permission. All rights reserved.)

20 -

14.2 IMAGE TRANSFORMATION AND LOW-LEVEL PROCESSING

In this section, we examine the first stages of processing. This includes the process of forming an image and transforming it to an array of numbers which can then be operated on by a computer. In this first stage, only local processing is performed on the numbers to reduce noise and other unwanted picture elements, and to accentuate object boundaries.

Transforming Light Energy to Numbers

The first step in image processing requires a transformation of light energy to numbers, the language of computers. To accomplish this, some form of light sensitive transducer is used such as a vidicon tube or charge-coupled device (CCD).

A vidicon tube is the type of sensor typically found in home or industrial video systems. A lens is used to project the image onto a flat surface of the vidicon. The tube surface is coated with a photoconductive material whose resistance is inversely proportional to the light intensity falling on it. An electron gun is used to produce a flying-spot scanner with which to rapidly scan the surface left to right and top to bottom. The scan results in a time varying voltage which is proportional to the scan spot image intensity. The continuously varying output voltage is then fed to an analog-to-digital converter (ADC) where the voltage amplitude is periodically sampled and converted to numbers. A typical ADC unit will produce 30 complete digitized frames consisting of 256 × 256, or 512 × 512 (or more) samples of an image per second. Each sample is a number (or triple of numbers in the case of color systems) ranging from 0 to 64 (six bits) or 0 to 255 (eight bits). The image conversion process is depicted in Figure 14.4.

A CCD is typical of the class of solid state sensor devices known as charge transfer devices that are now being used in many vision systems. A CCD is a rectangular chip consisting of an array of capacitive photodetectors, each capable of storing an electrostatic charge. The charges are scanned like a clock-driven shift register and converted into a time varying voltage which is proportional to the incident light intensity on the detectors. This voltage is sampled and converted to integers using an ADC unit as in the case of the vidicon tube. The density of the

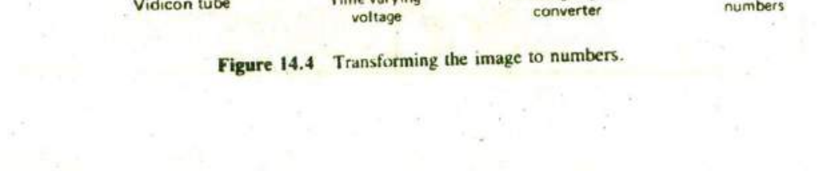


Figure 14.4 Transforming the image to numbers.

detectors on the chip is quite high. For example, a CCD chip of about five square centimeters in area may contain as many as 1000 by 1000 detectors.

The numeric outputs from the ADC units are collected as arrays of numbers which correspond to the light intensity of the image on the surface of the transducer. This is the input to the next stage of processing illustrated in Figure 14.4.

Processing the Quantized Arrays

The array of numbers produced from the image sensing device may be thought of as the lowest, most primitive level of abstraction in the vision understanding process. The next step in the processing hierarchy is to find some structure among the pixels such as pixel clusters which define object boundaries or regions within the image. Thus, it is necessary to transform the array of raw pixel data into regions of discontinuities and homogeneity, to find edges and other delimiters of these object regions.

A raw digitized image will contain some noise and distortion. Therefore, computations to reduce these effects may be necessary before locating edges and regions. Depending on the particular application, low level processing will often require local smoothing of the array to eliminate this noise. Other low level operations include threshold processing to help define homogeneous regions, and different forms of edge detection to define boundaries. We examine some of these low level methods next.

Thresholding is the process of transforming a gray level representation to a binary representation of the image. All digitized array values above some threshold level T are set equal to the maximum gray-level value (black), and values less than or equal to T are set equal to zero (white). For simplicity, assume gray-level values have been normalized to range between zero and one, and suppose a threshold level of $T = 0.7$ has been chosen. Then all array values $g(x,y) > 0.7$ are set equal to 1 and values $g(x,y) \leq 0.7$ are set equal to 0. The result is an array of binary 0 and 1 values. An example of an image that has been thresholded at 0.7 to produce a binary image is illustrated in Figure 14.5.

Thresholding is one way to segment the image into sharpen object regions by enhancing some portions and reducing others like noise and other unwanted features. Thresholding can also help to simplify subsequent processing steps. And in many cases, the use of several different threshold levels may be necessary since low intensity object surfaces will be lost to high threshold levels, and unwanted background will be picked up and enhanced by low threshold levels. Thresholding at several levels may be the best way to determine different regions in the image when it is necessary to compensate for variations in illumination or poor contrast.

Selecting one or more appropriate threshold level settings T_i will require additional computations, such as first producing a histogram of the image gray-level intensities. A histogram gives the frequencies of occurrence of different intensity (or some other feature) levels within the image. An analysis of a histogram can reveal where concentrations of different intensity levels occur, where peaks and broad flat levels occur and where abrupt differences in level occur. From this informa-

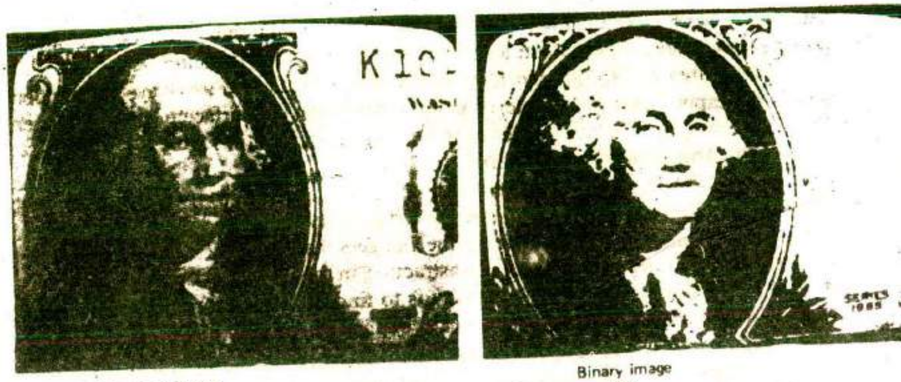


Figure 14.5 Threshold transformation of an image.[1]

tion the best choice of T_i values are often made apparent. For example, a histogram with two or more clear separations between intensity levels that have a relatively high frequency of occurrence will usually suggest the best threshold levels for object identification and separation. This is seen in Figure 14.6.

Next, we turn to the question of image smoothing. *Smoothing* is a form o digital filtering. It is used to reduce noise and other unwanted features and to enhanc certain image features. Smoothing is a form of image transformation that tends t eliminate spikes and flaten widely fluctuating intensity values. Various forms of smoothing techniques have been employed, including local averaging, the use of models, and parametric form fitting.

One common method of smoothing is to replace each pixel in an array with a weighted average of the pixel and its neighboring values. This can be accomplishe with the use of filter masks which use some configuration of neighboring pixe values to compute a smoothed replacement value. Two typical masks consist o either four or eight neighboring pixels whose intensity values are used in the weighting

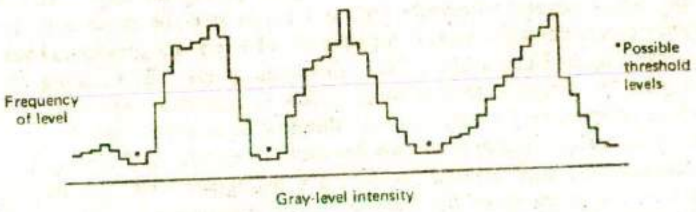


Figure 14.6 Histogram of light intensity levels.

[1] Courtesy of Kenneth Chapman and INTELLEDEX, INC.

computation. If smoothing is being performed at pixel location (x,y), the neighboring pixels are at the eight locations: $(x + 1,y - 1)$, $(x + 1,y)$, $(x + 1,y + 1)$, $(x,y + 1)$, $(x,y - 1)$, $(x - 1,y - 1)$, $(x - 1,y)$, and $(x - 1,y - 1)$. From these, either the four immediate neighbors (top, bottom, left, and right) or all eight neighbors are sometimes chosen.

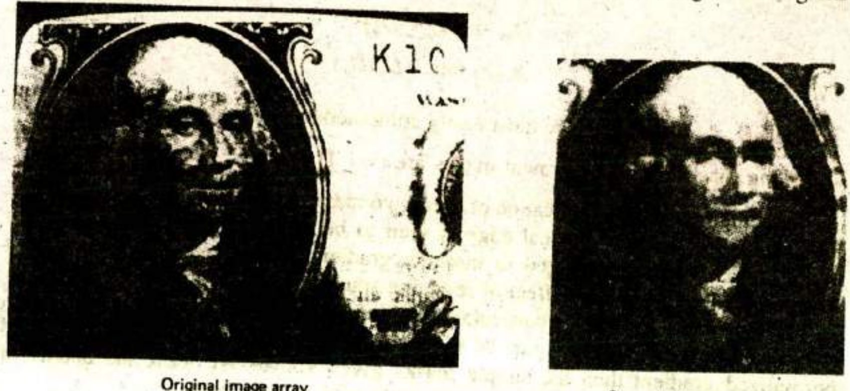
Examples of smoothing masks for four and eight neighborhood pixels are as follows:

$$\begin{bmatrix} 1/32 & 3/32 & 1/32 \\ 3/32 & \underline{1/2} & 3/32 \\ 1/32 & 3/32 & 1/32 \end{bmatrix} \qquad \begin{bmatrix} & 1/8 & \\ 1/8 & \underline{1/2} & 1/8 \\ & 1/8 & \end{bmatrix}$$

The underlined number in each array identifies the pixel being smoothed. (Note that the filter weights in a mask should sum to one to avoid distortions.) Applying a mask to an image array has the effect of reducing spurious noise as well as sharp boundaries. It reduces sharp spikes but also tends to blur the image. For example, when the eight mask filter given above is applied to the array shown in Figure 14.7, the blurring effects are quite pronounced.

The degree of smoothing and hence blurring can, of course, be controlled with the use of appropriate weighting values in the mask. Weighted smoothing of this type over a region is known as convolution. Convolution is sometimes used to smooth an image prior to the application of differential operators which detect edges. We return to the subject of convolution smoothing after we look at the edge detection problem.

Local edge detection is the process of finding a boundary or delimiter between two regions. An edge will show up as a relatively thin line or arc which appears as a measurable difference in contrast between two otherwise homogeneous regions.



Original image array
(a)

Smoothed image array
(b)

Figure 14.7 Application of a smoothing mask.[2]

[2] Courtesy of Kenneth Chapman and INTELLEDEX, INC.

Regions belonging to the same object are usually distinguishable by one or more features which are relatively homogeneous throughout, such as color, texture, three-dimensional flow effects, or intensity.

Boundaries which separate adjoining regions represent a discontinuity in one or more of these features, a fact that can be exploited by measuring the rate of change of a feature value over the image surface. For example, the rate of change or gradient in intensity in the horizontal and vertical directions can be measured with difference functions D_x and D_y defined as

$$D_x = f(x,y) - f(x - n,y)$$
$$D_y = f(x,y) - f(x,y - n)$$

where n is a small integer greater than or equal to 1.

When an image is scanned horizontally or vertically, D_x and D_y will vary little over homogeneous regions, and show a sharp increase or decrease at locations where discontinuities occur. They are the discrete equivalents of the continuous differential operators used in the calculus. The rate of change of the gradient can also be useful in finding local edges as we will see below. For discrete functions, second order difference operators provide the rate of change of gradient, comparable to second order differential operators.

Since we are interested in locating edges with any given orientation, a better gradient measure is one which is sensitive to intensity changes in any direction. We can achieve this with a directional norm of D_x and D_y such as the vector gradient.

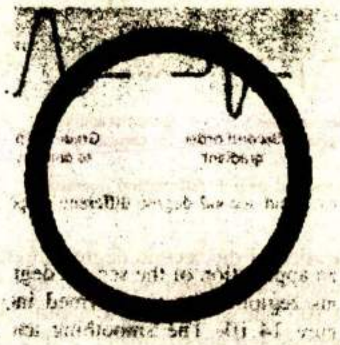
$$D_{xy} = (D_x^2 + D_y^2)^{1/2}$$
$$\theta_{xy} = \tan^{-1}(D_y/D_x)$$

For $n = 1$, D_x and D_y are most easily computed by application of the equivalent weighting masks; the two element masks are $(-1\ \underline{1})$ and $\left[\frac{1}{-1}\right]$ respectively.

An example of the application of these two masks to an image array is illustrated in Figure 14.8 where a vertical edge is seen to be quite pronounced. Masks such as these have been generalized to measure gradients over wider regions covering several pixels. This has the effect of reducing spurious noise and other sharp spikes.

Two masks deserving particular attention are the Prewitt (1970) and Sobel (1970) masks as depicted in Figure 14.9. These masks are used to compute a broadened normalized gradient than the simple masks given above. We leave the details of the computations as one of the exercises at the end of this chapter.

We return now to the methods of edge detection which employ smoothing followed by an application of the gradient. For this, the continuous case is considered first.


Original Array I
(a)

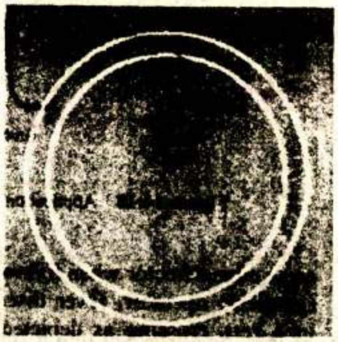

Result of Dx and Dy Mask Applied to I
(b)

Figure 14.8 Application of difference functions to an image.

The continuous analog of discrete smoothing in one dimension is the convolution of two functions f and g (written $f * g$) where

$$h(y) = f * g = \int f(x)g(y - x)dx$$

Convolving the two functions f and g is similar to computing the cross correlation, a process that reduces random noise and enhances coherent or structural changes.

One particular form of weighting function g has a symmetric bell shape or normal form, that is the Gaussian distribution. The two dimensional form of this function is given by

$$g(u,v) = ce^{-(u^2+v^2)/2}$$

where c is a normalizing constant.

Because of their rotational symmetry, Gaussian filters produce desirable effects

$$P_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & \underline{0} & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & \underline{0} & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$P_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & \underline{0} & 0 \\ -1 & -1 & -1 \end{bmatrix} \qquad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & \underline{0} & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Prewitt Masks Sobel Masks

Figure 14.9 Generalized edge detection masks.

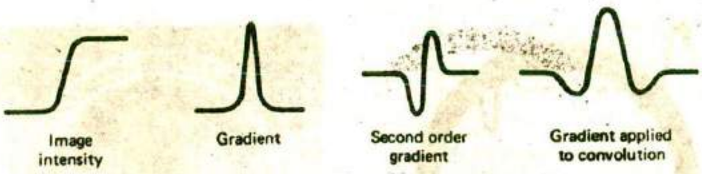


Figure 14.10 Application of Gaussian and second degree differential operators.

as an edge detector when followed by an application of the second degree differential (gradient) operator. Over discontinuous regions, the transformed intensity results in a zero crossing as depicted in Figure 14.10. The smoothing and differencing operations may be combined into a single operator and approximated with a digital mask of the types given above (Marr and Hildreth, 1980).

There is some psychological evidence to support the belief that the human eye uses a form of Gaussian transformation called lateral inhibition which has the effect of enhancing the contrast between gradually changing objects, such as an object and its background.

Another approach used to filter the digitized image applies frequency domain transforms such as the Fourier transform. Since edges represent higher frequency components, the transformed image can be analyzed on the basis of its frequency distribution. For this, the Fourier transform has become one of the most popular transform methods, since an efficient computation algorithm has been developed. It is known as the Fast Fourier transform. The discrete two-dimensional version of the Fourier transform is given by

$$F(u,v) = \frac{1}{n} \sum_{x=0}^{n-1} \sum_{y=0}^{n-1} f(x,y)\, e^{-2i(xu+yv)/n}$$

Applying this transform to an array of intensity values produces an array of complex numbers that correspond to the spatial frequency components of the image (sums of sine and cosine terms). The transformed array will contain all of the information in the original intensity image, but in a form that is more easily used to identify regions that contain different frequency components. Filtering with the Fourier transform is accomplished by setting the high (or low) values of u and v equal to zero. For example, the value $F(v,v) = F(0,0)$ corresponds to the zero frequency or the DC component, and higher values of u and v correspond to the high frequency components. As with intensity image arrays, thresholding of transformed arrays can be used to separate different frequency components.

The original intensity image with any modifications, is recovered with the inverse transform given by

$$f(x,y) = \frac{1}{n} \sum_{u=0}^{n-1} \sum_{v=0}^{n-1} F(u,v) \exp\left[\frac{2i}{n}(xu + yv)\right]$$

Another method of edge detection is model fitting. This is accomplished by locally fitting a parametric profile of edges to the image array. A model in the form of a mask is shifted over a region and compared to the corresponding gray levels. If the fit between the model and the gray-level pattern scores high enough, an edge with the given orientation is labeled appropriately. Model fitting methods usually require heavy computations. We omit the details here.

Texture and Color

As suggested earlier, texture and color are also used to identify regions and boundaries. Texture is a repeated pattern of elementary shapes occurring on an object's surface. Texture may appear to be regular and periodic, random, or partially periodic. Figure 14.11 illustrates some examples of textured surfaces.

The structure in texture is usually too fine to be resolved, yet still course enough to cause noticeable variation in the gray levels. Even so, methods of analysis for texture have been developed. They are commonly based on statistical analyses of small groups of pixels, the application of pattern matching, the use of Fourier transforms, or modeling with special functions known as fractals. These methods are beyond the scope of our goals in this chapter.

The use of color to identify and interpret regions requires more than three times as much processing as gray-level processing. First, the image must be separated into its three primary colors with red, green, and blue filters (Figure 14.12).

The separate color images must then be processed by sampling the intensities and producing three arrays or a single array of tristimulus values. The arrays are then processed separately (in some cases jointly) to determine common color regions and corresponding boundaries. The processes used to find boundaries and regions, and to interpret color images is similar to that of gray-level systems.

Although the additional computation required in color analysis can be significant, the added information gained from separate color intensity arrays may be warranted, depending on the application. In complex scene analysis, color may be the most effective method of segmentation and object identification. In Section 14.6 we describe

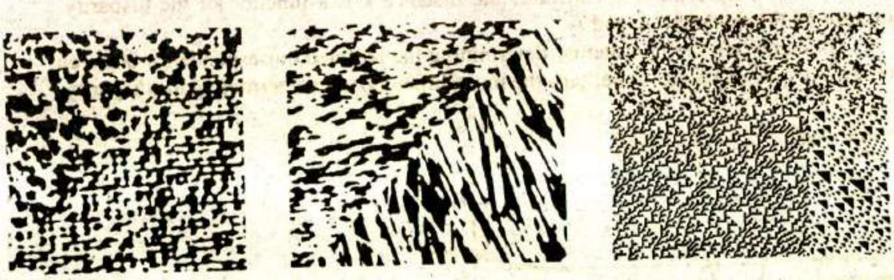


Figure 14.11 Examples of textured surfaces.

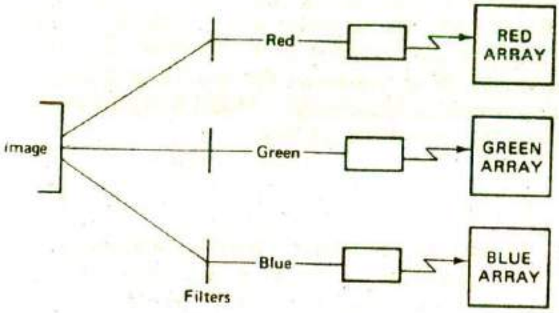


Figure 14.12 Color separation and processing.

an interesting color scene analyser which is based on a rule based inferencing system (Ohta, 1985).

Stereo and Optic Flow

A stereoscopic vision system requires two displaced sensors to obtain two views of objects from different perspectives. The differences between the views makes it possible to estimate distances and derive a three-dimensional model of a scene. The displacement of a pixel from one image to a different location in another image is known as the disparity. It is the dispatity between the two views that permit the estimation of the distance to objects in the scene. The human vision system is somehow able to relate the two different images and form a correspondence that translates to a three-dimensional interpretation. Figure 14.13 illustrates the geometric relationships used to estimate distances to objects in stereoscopic systems.

The distance k from the lens to the object can be estimated from the relationships that hold between the sides of the similar triangles. Using the relations $i_1 / e_1 = f / k$, $i_2 / e_2 = f / k$, and $d = e_1 + e_2$ we can write

$$k = fd / (i_1 + i_2)$$

Since f and d are relatively constant, the distance k is a function of the disparity, or sum of the distances i_1 and i_2.

In computer vision systems, determining the required correspondence between the two displaced images is perhaps the most difficult part in determining the disparity.

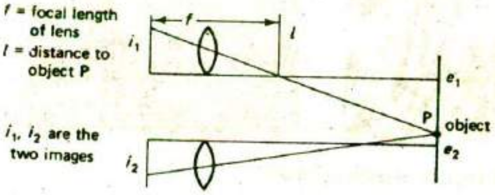


Figure 14.13 Disparity in stereoscopic systems.

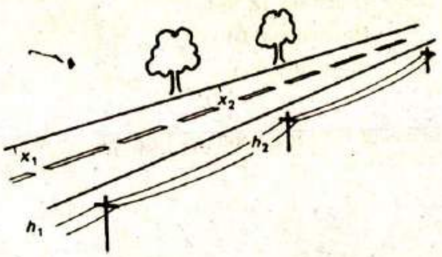


Figure 14.14 Example of optical flow in a scene.

Corresponding pixel groupings in the two images must be located to determine the disparity from which the distance can be estimated. In practice, methods based on correlation, gray-level matching, template matching, and edge contour comparisons have been used to estimate the disparity between stereo images.

Optic flow is an alternative approach to three-dimensional scene analysis which is based on the relative motion of a sensor and objects in the scene. If a sensor is moving (or objects are moving past a sensor), the apparent continuous flow of the objects relative to the sensor is known as optical flow. Distances can be estimated from the change in flow or relative velocity of the sensor and the objects. For example, in Figure 14.14 if the velocity of the sensor is constant, the change in distance dx between points x_1 and x_2 is proportional to the change in size of the power lines h, through the relation

$$dx / dt = k(dh / dt)$$

This relationship is equivalent to the change in size of regular flowing objects with distance from the observer such as highways, railroad tracks, or power lines, as depicted in Figure 14.14.

14.3 INTERMEDIATE-LEVEL IMAGE PROCESSING

The next major level of analysis builds on the low-level or early processing steps described above. It concentrates on segmenting the image space into larger global structures using homogeneous features in pixel regions and boundaries formed from pieces of edges discovered during the low-level processing. This level requires that pieces of edges be combined into contiguous contours which form the outline of objects, partitioning the image into coherent regions, developing models of the segmented objects, and then assigning labels which characterize the object regions.

One way to begin defining a set of objects is to draw a silhouette or sketch of their outlines. Such a sketch has been called the raw primal sketch by Marr (1982). It requires connecting up pieces of edges which have a high likelihood of forming a continuous boundary. For example, the problem is to decide whether two edge pieces such as

```
(edge (location 21 103)      (edge (location 18 98)
      (intensity 0.8)              (intensity 0.6)
      (direction 46))              (direction 41))
```

should be connected. This general process of forming contours from pieces of edges is called *segmentation*.

Graphical Edge Finding

Graphical methods can be used to link up pieces of edges. One approach is to use a minimum spanning tree (MST). Starting at any cluster of pixels known to be part of an edge, this method performs a search in the neighborhood of the cluster for groupings with similar feature values. Each such grouping corresponds to a node in an edge tree. When a number of such nodes have been found they are connected using the MST algorithm.

An MST is found by connecting an arc between the first node selected and its closest neighbor node and labeling the two nodes accordingly. Neighborhoods of both connected nodes are then searched. Any node found closest to either of the two connected nodes (below some threshold distance) is then used to form the next branch in the tree. A second arc is constructed between the newly found node and the closest connected node, again labeling the new node. This process is repeated until all nodes having arc distances less than some value (such as a function of the average arc distances) have been connected. An example of an MST is given in Figure 14.15.

Another graphical approach is based on the assignment of a cost or other measure of merit to pixel groupings. The cost assignment can be based on a simple function of features, such as intensity, orientation, or color. A best-first (branch-and-bound) or other form of graph search is then performed using some heuristic function to determine a least-cost path which represents the edge contour.

Other edge finding approaches are based on fitting a low degree polynomial to a number of edge pieces which have been found through local searches. The resultant polynomial curve segment is then taken as the edge boundary. This approach is similar to one which compares edge templates to short groupings of pieces. If a particular matching template scores above some threshold, the template pattern is then used to define the contour.

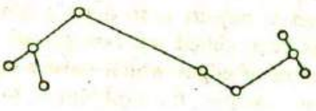


Figure 14.15 Edge finding using a minimum spanning tree.

Edge Finding with Dynamic Programming

Edge following can also be formulated as a dynamic programming problem since picking a best edge path is a type of sequential optimizion problem. Candidate edge pieces are assigned a local cost value based on some feature such as intensity, and the path having minimum cost is defined as the edge contour.

Assume that a starting point E_s has been selected. Dynamic programming begins with a portion of the problem and finds an optimal solution for this subproblem. The subproblem is then enlarged, and an optimal solution is found for the enlarged problem. This process continues step-by-step until a global optimum for the whole problem has been found (a path to the terminal edge point E_t).

The process can be described mathematically as a recursive process. Let $C_n(s,t_n)$ be the total cost of the best path for the remaining path increments, given the search is at position (state) s and ready to select t_n as the next move direction. Let $t_n{}^*$ be the value of t_n that minimizes C_n, and $C_n{}^*$ the corresponding minimum of C_n. Thus, at each stage, the following values are computed:

$$C_n{}^*(s) = \min_{t_n} C_n(s,t_n) = C_n(s,t_n{}^*)$$

where

$$\begin{aligned} C_n(s,t_n) &= (\text{cost at stage } n) \\ &\quad + (\text{minimum costs for stages } n+1 \text{ onwards}) \\ &= K(s,t_n) + C_{n+1}{}^*(t_n) \end{aligned}$$

where $K(s,t_n)$ is the cost at stage n, and $C_{n+1}{}^*(t_n)$ is the minimum cost for stages $n + 1$ to the terminal stage.

The computation process is best understood through an example. Consider the following 5×5 array of pixel cost values.

$$\begin{bmatrix} 9 & 7 & 6 & 5 & 1 \\ 3 & 7 & 2 & 7 & 1 \\ 4 & 1 & 5 & 2 & 7 \\ 6 & 6 & 3 & 7 & 7 \\ 8 & 7 & 2 & 2 & 3 \end{bmatrix}$$

Suppose we wish to find the optimal cost path from the lower left to the upper right corner of the array. We could work from either direction, but we arbitrarily choose to work forward from the lower left pixel with cost value 8. We first set all values except 8 equal to some very large number, say M, and compute the minimum cost of moving from the position with the 8 to all other pixels in the bottom row by adding the cost of moving from pixel to neighboring pixel. This results in the following cost array.

$$\begin{bmatrix} M & M & M & M & M \\ M & M & M & M & M \\ M & M & M & M & M \\ M & M & M & M & M \\ 8 & 15 & 17 & 19 & 22 \end{bmatrix}$$

Next, we compute the minimum neighbor path cost for the next to the last row to obtain

$$\begin{bmatrix} M & M & M & M & M \\ M & M & M & M & M \\ M & M & M & M & M \\ 14 & 14 & 17 & 24 & 29 \\ 8 & 15 & 17 & 19 & 22 \end{bmatrix}$$

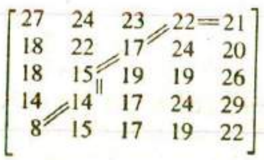
Note that the minimum cost path to the second, third and fourth positions in this row is the diagonal path (position 5,1 to 4,2) followed by a horizontal right traversal in the same row, whereas the minimum cost path for the last position in this row is the path passing through the rightmost position of the bottom row. The remaining minimum path costs are computed in a similar fashion, row by row, to obtain the final cost array.

$$\begin{bmatrix} 27 & 24 & 23 & 22 & 21 \\ 18 & 22 & 17 & 24 & 20 \\ 18 & 15 & 19 & 19 & 26 \\ 14 & 14 & 17 & 24 & 29 \\ 8 & 15 & 17 & 19 & 22 \end{bmatrix}$$

From this final minimum cost array, the least cost path is easily found to be

$$(5,1) \rightarrow (4,1) \text{ or } (4,2) \rightarrow (3,2) \rightarrow (2,3) \rightarrow (1,4) \rightarrow (1,5)$$

as depicted with the double line path.



Dynamic programming methods usually result in considerable savings over exhaustive search methods which require an exponential number of computations and comparisons (see Chapter 9), whereas the number of dynamic programming computations for the same size of problem is of linear order.

Region Segmentation through Splitting and Merging

Rather than defining regions with edges, it is possible to build them. For example, global structures can be constructed from groups of pixels by locating, connecting, and defining regions having homogeneous features such as color, texture, or intensity. The resulting segmented regions are expected to correspond to surfaces of objects in the real world. Such coherent regions do not always correspond to meaningful regions, but they do offer another viable approach to the segmentation of an image. When these methods are combined with other segmentation techniques, the confidence level that the regions represent meaningful objects will be high.

Once an image has been segmented into disjointed object areas, the areas can be labeled with their properties and their relationships to other objects, and then identified through model matching or description satisfaction.

Region segmentation may be accomplished by region splitting, by region growing (also called region merging), or by a combination of the two. When splitting is used, the process proceeds in a top-down manner. The image is split successively into smaller and smaller homogeneous pieces until some criteria are satisfied. When growing regions, the process proceeds in a bottom-up fashion. Individual pixels or small groups of pixels are successively merged into contiguous, homogeneous areas. A combined splitting-growing approach will use both bottom-up and top-down techniques.

Regions are usually assumed to be disjointed entities which partition the image such that (1) a given pixel can appear in a single region only, (2) subregions are composed of connected pixels, (3) different regions are disjoint areas, and (4) the complete image area is given by the union of all regions. Regions are usually defined by some homogeneous property such that all pixels belonging to the region satisfy the property, and pixels not satisfying the property lie in a different region. Note that a region need not consist of contiguous pixels only since some objects may be split or covered by occluding surfaces. Condition 2 is needed to insure that all regions are accounted for and that they fill up the complete image area.

In region splitting, the process begins with an entire image which is successively divided into smaller regions which exhibit some coherence in features. One effective method is to apply multiple thresholding levels which can isolate regions having homogeneous features. Histograms are first obtained to establish the threshold levels. This may require masking portions of the image to achieve effective separation of complex objects. Each threshold level can then produce a binary image consisting of all of the objects which exceed the thresholded level. Once the binary regions are formed, they are easily delineated, separated, and marked for subsequent processing. This whole process of masking, computing, and analyzing a histogram, thresholding, defining an area, masking, and so on can be performed in a recursive manner. The process terminates when the masks produce monomodal histograms with the image fully partitioned.

Segmentation techniques based on region growing start with small atomic regions (one or a few pixels) and build coherent pixel regions in a bottom-up fashion.

Local features such as the intensity of a group of pixels relative to the average intensity of neighboring pixels are used as criteria for the merging operation. A low level of contrast between contiguous groups gives rise to the merging of areas, while a higher level of contrast, such as found at boundaries, provides the criteria for region segregation.

Split-and-merge techniques attempt to gain the advantages of both methods. They combine top-down and bottom-up processing using both region splitting and merging until some split-merge criterion no longer exists. At each step in the process, split and merge threshold values can be compared and the appropriate operation performed. In this way, over-splitting and under-merging can be avoided.

14.4 DESCRIBING AND LABELING OBJECTS

We continue in this section with further intermediate-level processing steps all aimed at building higher levels of abstraction. The processing steps here are related to describing and labeling the regions.

Once the image has been segmented into disjointed regions, their shapes, spatial interrelationships, and other characteristics can be described and labeled for subsequent interpretation. This process requires that the outlines or boundaries, vertices, and surfaces of the objects be described in some way. It should be noted, however, that a description for a region can be based on a two- or three-dimensional image interpretation. Initially, we focus on the two-dimensional interpretation.

Typically, a region description will include attributes related to size, shape, and general appearance. For example, some or all of the following features might be included.

Region area
Contour length (perimeter) and orientation
Location of the center of mass
Minimum bounding rectangle
Compactness (area divided by perimeter squared)
Fitted scatter matrix of pixels
Number and characteristics of holes or internal occlusions
Minimum bounding rectangle
Degree and type of texture
Average intensity (or average intensities of base colors)
Type of boundary segments (sharp, fuzzy, and so on) and their locations
Boundary contrast
Chain code (described below)
Shape classification number (task specific)
Position and types of vertices (number of adjoining segments)

Some of the above features are illustrated in Figure 14.16.

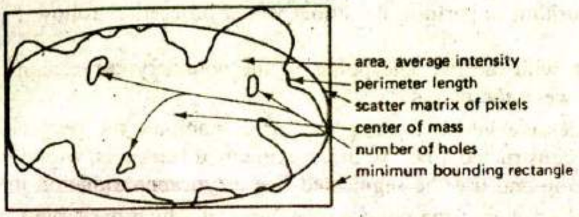


Figure 14.16 Descriptive features for a region.

In addition to these characteristics, the relationships between regions may also be important, particularly adjacent regions. Relations between regions can include their relative positions and orientations, distances between boundaries, intervening regions, relative intensities or contrasts in color, degree of abutment, and degree of connectivity or concentration. When the image domain is known, domain or task specific features may also be useful.

Next, we examine some of the definitions and methods used for region descriptions.

Describing Boundaries

Boundaries can be described as linked straight-line segments, fitted polynomial curves, or in a number of other ways. One simple method of fitting straight-line segments to an arbitrary boundary is by successive linear segmentation fitting. This method permits any degree of fit accuracy at the expense of computation time. The fitting procedure is illustrated in Figure 14.17.

The fitting begins by connecting a single straight line to the two end points and using this as an approximation to the curve (a). Additional lines are then constructed using the points on the curve at maximum perpendicular distances to the fitted lines (b, c, and d).

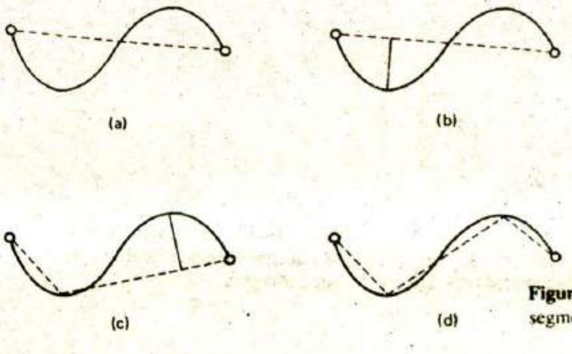


Figure 14.17 Curve fitting with linear segments.

An algorithm to perform the fitting would proceed as follows.

1. Starting with the two end points of the boundary curve, construct a straight line between the points.
2. At successive intervals along the curve, compute the perpendicular distance to the constructed line. If the maximum distance is within some specified limit, stop and use the segmented line as an approximation to the boundary.
3. Otherwise, choose the point on the curve at which the largest distance occurs and use this as a breakpoint with which to construct two new line segments which connect to the two endpoints. Continue the process recursively with each subcurve until the stopping condition of Step 2 is satisfied.

Chain Codes

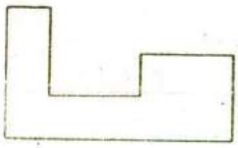
Another method used for boundary descriptions is known as chain coding. A *chain code* is a sequence of integers which describe the boundary of a region in terms of displacements from some starting point. A chain code is specified by four or more direction numbers which give a trace of the directional displacements of successive unit line segments. An example of a four direction chain code is presented in Figure 14.18.

Chain code descriptions are useful for certain types of object matchings. If the starting position is ignored, the chain code is independent of object location. A derivative or difference (mod 4) for a chain code can also be useful since it is invariant under object rotation. The derivative is found by counting the number of 90 degree counterclockwise rotations made from segment to segment. Thus, the derivative for the chain code of Figure 14.18 is the code

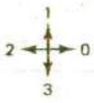
10000303001000103000300003000000000.

Other Descriptive Features

Some other descriptive features include the area, intensity, orientation, center of mass, and bounding rectangle. These descriptions are determined in the following way.



(a)



(b)

(c) Chain code: 1111100333000011000033333222222222

Figure 14.18 (a) Region boundary, (b) direction numbers, and (c) chain code for region boundary.

1. The area of a region can be given by a count of the number of pixels contained in the region.
2. The average region intensity is just the average gray-level intensity taken over all pixels in the region. If color is used in the image, the average is given as the three base color intensity averages.
3. The center of mass M_c for a region can be computed as the average x-y vector position (denoted as P_i), that is

$$M_c = (1/n) \sum_{i=1}^{n} P_i$$

4. The scatter matrix S_m is an elliptical area which approximates the shape of the region. It may be computed as the average distance from the center of the region mass as follows.

$$S_m = \sum_{i=1}^{n} (P_i - M_c)(P_i - M_c)^t$$

where t denotes matrix transposition.

5. The minimum bounding rectangle is found as the rectangular area defined by the intersection of the horizontal and vertical lines which pass through the maximum and minimum pixel positions of the region.

Three-Dimensional Descriptions

Up to this point, we have been mainly concerned with developing two-dimensional descriptions for images. But for many applications, an analysis which produces a three-dimensional scene description will be required. When a stereo system is being used, the methods described in the previous section for stereo analysis can be applied to estimate such parameters as depths, volumes, and distances of objects. When a two-dimensional image is being used as the source, this information must be determined by other means.

Several programs capable of interpreting images consisting of three-dimensional polyhedral blocks world objects were written beginning in the early 1960s (Roberts, 1965, Guzman, 1969, Huffman, 1971, Clowes, 1971 and Waltz, 1975). The experience gained from this work has led to algorithms and techniques with which to classify and identify regular complex polyhedral types of objects from two-dimensional images.

Roberts wrote a program which began by finding lines in the image which corresponded to the edges of the polyhedral objects. It then used descriptions of the lines to match against stored models of primitive objects such as wedges, cubes, and prisms. To perform the match operation, it was necessary to transform the objects by scaling and performing rotations and translations until a best match was possible. Once a match was obtained and all objects identified, the program demonstrated its "understanding" of the scene by producing a graphic display of it on a monitor screen.

Guzman wrote a program called SEE which examined how surfaces from the same object were linked together. The geometric relationships between different types of line junctions (vertices) helped to determine the object types. Guzman identified eight commonly occurring edge junctions for his three-dimensional blocks world objects. The junctions were used by heuristic rules in his program to classify the different object by type (Figure 14.19).

Huffman and Clowes, working independently, extended this work by developing a line labeling scheme which systematized the classification of polyhedral objects. Their scheme was used to classify edges as either concave, convex, or occluding. Concave edges are produced by two adjacent touching surfaces which produce a concave (less than 180°) depth change. Conversely, convex edges produce a convexly viewed depth change (greater than 180°), and an occluding edge outlines a surface that obstructs other objects.

To label a concave edge, a minus sign is used. Convex edges are labeled with a plus sign, and a right or left arrow is used to label the occluding or boundary edges. By restricting vertices to be the intersection of three object faces (trihedral vertices), it is possible to reduce the number of basic vertex types to only four: the L, the T, the Fork, and the Arrow (Figure 14.20). Different label combinations assigned to these four types then assist in the classification and identification of objects.

When a three-dimensional object is viewed from all possible positions, the four junction types, together with the valid edge labels, give rise to eighteen different permissible junction configurations as depicted in Figure 14.20. From a dictionary of these valid junction types, a program can classify objects by the sequence of bounding vertices which describe it. Impossible object configurations such as the one illustrated in Figure 14.21 can also be detected.

Geometric constraints, together with a consistent labeling scheme, can greatly simplify the object identification process. A set of labeling rules which greatly facilitates this process can be developed for different classes of objects. For example, using the labels described above, the following rules will apply for many polyhedral

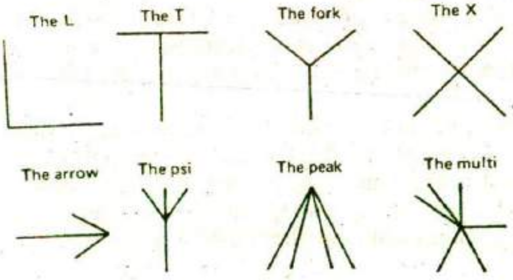


Figure 14.19 Three-dimensional polyhedral junction types.

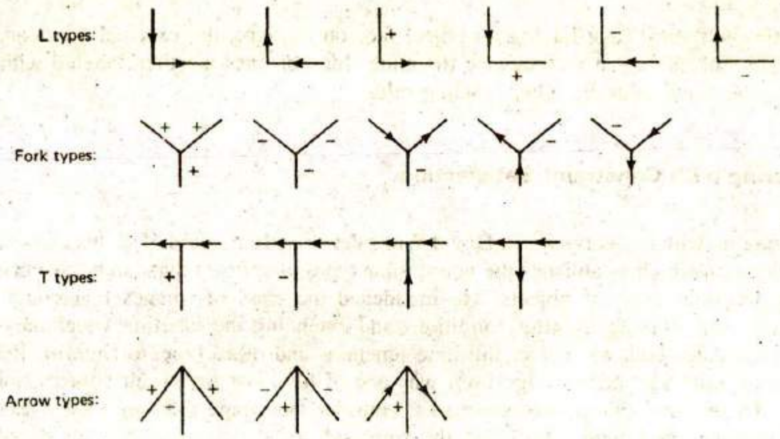


Figure 14.20 Valid junction labels for three-dimensional shapes.

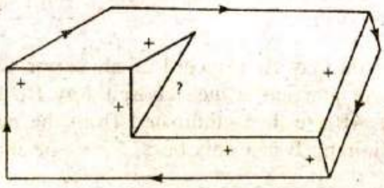


Figure 14.21 Example of an impossible object.

objects: (1) the arrow should be directed to mark boundaries by traversing the object in a clockwise direction (the object face appears on the right of the arrow), (2) unbroken lines should have the same label assigned at both ends, (3) when a fork is labeled with a + edge, it must have all three edges labeled as +, and (4) arrow junctions which have a → label on both barb edges must also have a + label on the shaft.

These rules can be applied to a polygonal object as illustrated in Figure 14.22.

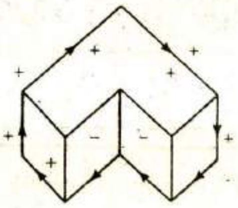


Figure 14.22 Example of object labeling.

Starting with any edge having an object face on its right, the external boundary is labeled with the → in a clockwise direction. Interior lines are then labeled with + or − consistent with the other labeling rules.

Filtering with Constraint Satisfaction

Continuing with this early work, David Waltz developed a method of vertex constraint propagation which establishes the permissible types of vertices that can be associated with a certain class of objects. He broadened the class of images that could be analyzed by relaxing lighting conditions and extending the labeling vocabulary to accommodate shadows, some multiline junctions and other types of interior lines. His constraint satisfaction algorithm was one of his most important contributions.

To see how this procedure works, consider the image drawing of a pyramid as illustrated in Figure 14.23. At the right side of the pyramid are all possible labelings for the four junctions A, B, C, and D.

Using these labels as mutual constraints on connected junctions, permissible labels for the whole pyrimid can be determined. The constraint satisfaction procedure works as follows:

1. Starting at an arbitrary junction, say A, a record of all permissible labels is made for that junction. An adjacent junction is then chosen, say B, and labels which are inconsistent with the line AB are then eliminated from the permissible A and B lists. In this case, the line joining B can only be a +, −, or an up-arrow

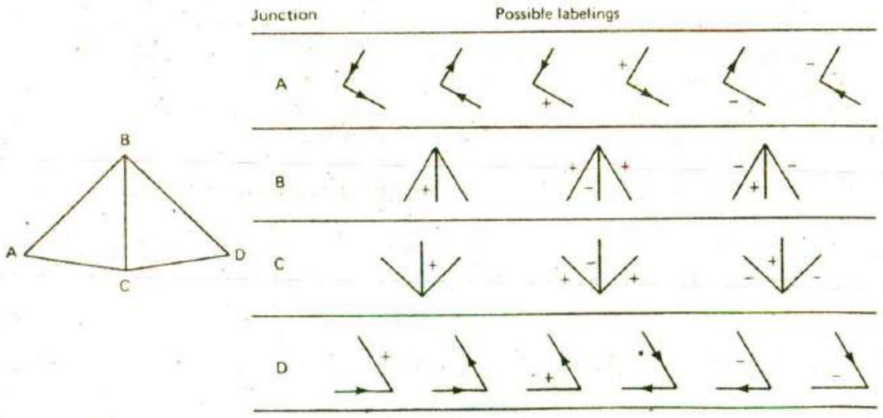
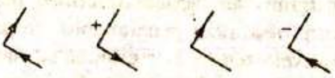
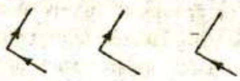


Figure 14.23 Possible labelings for an object.

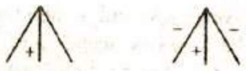
→. Consequently, two of the possible A labelings can be eliminated with the remaining four being



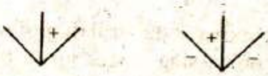
2. Choosing junction C next, we find that the BC constraints are satisfied by all of the B and C labelings; so no reduction is possible with this step. On the other hand, the line AC must be labeled as − or as an up-left-arrow ← to be consistent. Therefore, an additional label for A can be eliminated to reduce the remainder to the following.



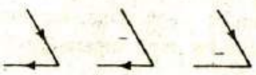
3. This new restriction on A now permits the elimination of one B labeling to maintain consistency. Thus, the permissible B labelings remaining are now



This reduction in turn, places a new restriction on BC, permitting the elimination of one C label, since BC must now be labeled as a + only. This leaves the remaining C labels as



4. Moving now to junction D, we see that of the six possible D labelings, only three satisfy the BD constraint of a − or a down-arrow. Therefore, the remaining permissible labelings for D are now



Continuing with the above procedure, it will be found that further label eliminations are not possible since all constraints have been satisfied. The above process is completed by finding the different combinations of unique labelings that can be assigned to the figure. This can be accomplished through a tree search process. A simple enumeration of the remaining labels shows that it is possible to find only three different labelings. We leave the labeling as an exercise at the end of this chapter.

The process of constraint satisfaction described here has been called Waltz filtering. It is a special form of a more general solution process called relaxation in which constraints are iteratively reduced or eliminated until equilibrium is reached. Such processes can be very effective in reducing large search spaces to manageable ones.

Template Matching

Template matching is the process of comparing patterns found in an image with prestored templates that are already named. The matching process may occur at lower levels using individual or groups of pixels using correlation techniques or at higher image processing levels using labeled region structures. Comparisons between object and template may be based on an exact or on a partial match; and, the matching process may use whole or component pieces when comparing the two. Rigid or flexible templates may also be used. (An example of flexible template matching was described in Section 10.5, Partial Matching, where the idea of a rubber mask template was introduced.)

Template matching can be effective only when the search process is constrained in some way. For example, the types of scenes and permissible objects should be known in advance, thereby limiting the possible pattern-template pairs. The use of some form of informed search can help restrict the size of the search space.

HIGH-LEVEL PROCESSING

Before proceeding with a discussion of the final (high-level) steps in vision processing, we shall briefly review the processing stages up to this point. We began with an image of gray-level or tristimulus color intensity values and digitized this image to obtain an array of numerical pixel values. Next, we used masks or some other transform (such as Fourier) to perform smoothing and edge enhancement operations to reduce the effects of noise and other unwanted features. This was followed by edge detection to outline and segment the image into coherent regions. The product of this step is a primal sketch of the objects. Region splitting and/or merging, the dual of edge finding, can also be used separately or jointly with edge finding as part of the segmentation process.

Histogram computations of intensity values and subsequent analyses were an important part of the segmentation process. They help to establish threshold levels which serve as cues for object separation. Other techniques such as minimum spanning tree or dynamic programming are sometimes used in these early processing stages to aid in edge finding.

Following the segmentation process, regions are analyzed and labeled with their characteristic features. The results of these final steps in intermediate-level processing is a set of region descriptions (data structures). Such structures are used as the input to the final high-level image processing stage. A summary of the data structures produced from the lowest processing stage to the final interpretation stage then can be depicted as follows.

$$\begin{tikzcd}
\text{Scene} \\
\text{Objects} \arrow[u] \\
\text{Regions} \arrow[u] \\
\text{Edges or subregions} \arrow[u] \\
\text{Pixels} \arrow[u]
\end{tikzcd}$$

Marr's Theory of Vision

David Marr and his colleagues (1982, 1980, and 1978) proposed a theory of vision which emphasized the importance of the representational scheme used at each stage of the processing. His proposal was based on the assumption that processing would be carried out in several steps similar to the summary description given above. The steps, and the corresponding representations are summarized as follows.

1. Gray-level image. The lowest level in processing consists of the two-dimensional array of pixel intensity levels. These levels correspond to important physical properties of world objects, the illumination, orientation with respect to the observer, geometry, surface reflectances, and object discontinuities. Processing at this stage is local with structure being implicit only. The key aspect of representation at this level is that it facilitate local and first-order statistical transformations.

2. Raw primal sketch. The primal sketch is a two-dimensional representation which makes object features more iconic and explicit. It consists of patterns of edge segments, intensity changes, and local features such as texture. The representation at this stage should emphasize pictorial image descriptions and facilitate transformations to the next stage where surface features and volumes are described.

3. The $2\frac{1}{2}$-dimensional sketch. This sketch is an explicit representation of a three-dimensional scene in which objects are given a viewer-centered coordinate system. The models here provide distance, volume, and surface structure. Three-dimensional spatial reconstruction requires the use of various tools including stereopsis, shape contours, shape from shading and texture, and other tools described earlier.

4. The three-dimensional models. The representations at the three-dimensional model level are symbolic ones giving attribute, relational, and geometric descriptions of the scene. The use of generalized cones and cylinders help to represent many object types, and hierarchical descriptions facilitate processing.

High-Level Processing

High-level processing techniques are less mechanical than either of the preceeding image processing levels. They are more closely related to classical AI symbolic methods. In the high-level processing stage, the intermediate-level region descriptions are transformed into high-level scene descriptions in one of the knowledge representation formalisms described earlier in Part II (associative nets, frames, FOPL statements, and so on; see Figure 14.24).

The end objective of this stage is to create high-level knowledge structures which can be used by an inference program. Needless to say, the resulting structures should uniquely and accurately describe the important objects in an image including their interrelationships. In this regard, the particular vision application will dictate the appropriate level of detail, and what is considered to be important in a scene description.

There are various approaches to the scene description problem. At one extreme, it will be sufficient to simply apply pattern recognition methods to classify certain objects within a scene. This approach may require no more than application of the methods described in the preceding chapter. At the other extreme, it may be desirable to produce a detailed description of some general scene and provide an interpretation of the function, purpose, intent, and expectations of the objects in the scene. Although this requirement is beyond the current state-of-the-art, we can say that it will require a great many prestored pattern descriptions and much general world knowledge. It will also require improvements on many of the processing techniques described in this chapter.

```
(region6
   (mass-center 23 48)
   (shape-code 24)
   (area 245)
   (number-boundary-segments 6)
   (chain-code 1133300011. . .)
   (orientation 85)
   (borders (region4 (position left-of) (contrast 5))
            (region7 (position above) (contrast 2))
               . . .
   (mean-intensity 0.6)
   (texture light regular)
```

Figure 14.24 Typical description of a segmented region.

Before a scene can be described in terms of high-level structures, prestored model descriptions of the objects must be available. These descriptions must be compared with the region descriptions created during the intermediate-level stage. The matching process can take the form of rule instantiations, segmented graph or network matchings, frame instantiations, traversal of a discrimination network (decision tree), or even response to message patterns in an object oriented system. The type of matching will naturally be influenced by the representation scheme chosen for the final structures.

To round out this section, we consider some of the approaches used in the high-level processing stage. In the following section, we consider some complete vision system architectures.

Associative networks have become a popular representation scheme for scene descriptions since they show the relationships among the objects as well as object characteristics. A simple example of an outdoor scene representation is illustrated in Figure 14.25.

Scene descriptions such as this can be formulated by interpreting region descriptions of the type shown in Figure 14.16. The interpretation knowledge will be encoded in production rules or other representation scheme. For example, a rule used to identify a sky region in an outdoor color scene would be instantiated with sky properties such as intensity, color, shape, and so forth. A rule to identify houses in an aerial photograph would be instantiated with conditions of area, compactness, texture, type of border, and so on, as illustrated in Figure 14.26. Rule conditions will sometimes have fuzzy or probability predicates to allow for similarity or partial

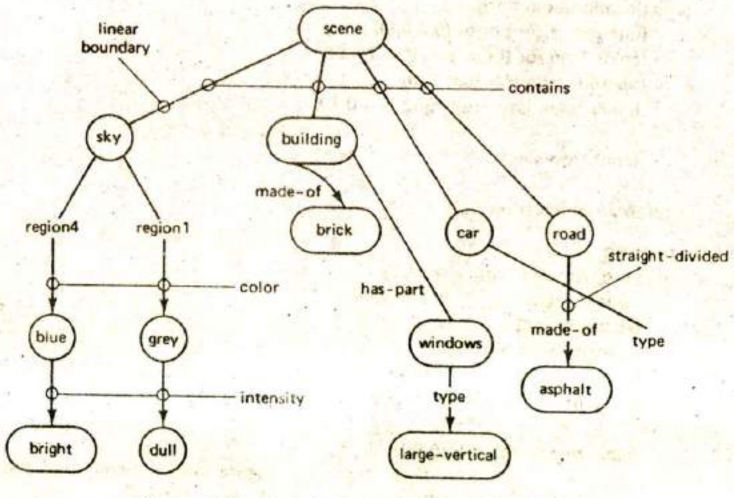


Figure 14.25 Associative network scene representation.

matches rather than absolute ones. Rule conclusions will be rated by likelihood or certainty factors instead of complete certainty. Identification of objects can then be made on the basis of a likelihood score. In Figure 14.26 (a) pairs of numbers are given in the antecedent to suggest acceptable condition levels comparable to Dempster-Shafer probabilities (the values in the figure are arbitrarily chosen with a scale of 0 to 1.0)

When rule-based identification is used, the vision system may be given an initial goal of identifying each region. This can be accomplished with a high-level goal statement of the following type.

```
(label region
   (or (*rgn = building)
       (*rgn = bushes)
       (*rgn = car)
       (*rgn = house)
       (*rgn = road)
       (*rgn = shadow)
       (*rgn = tree)))
```

Other forms of matching may also be used in the interpretation process. For example, a decision tree may be used in which region attributes and relation values determine the branch taken at each node when descending the tree. The leaves of the decision tree are labeled with the object identities as in Figure 14.27.

```
(R10-sky
   (and (location upper *rgn)
        (intensity *rgn bright (0.4 0.8))
        (color *rgn (or (blue grey)) (0.7 1.0))
        (textural *rgn low (0.8 1.0))
        (linear-boundary *rgn rgn2 (0.4 0.7)))
   →
        (label *rgn sky))
```

(a) Sky Identification Rule

```
(R32-building
   (and (intensity-avg *rgn > image)
        (area >= 60)
        (area <= 250)
        (compactness >= 0.6)
        (texture-variation <= 64.0)
        (percent border-planer >= 60)
   →
        (label region HOUSE (0.9))))
```

(b) Building Identification Rule

Figure 14.26 Interpretation rules for a sky and a building.

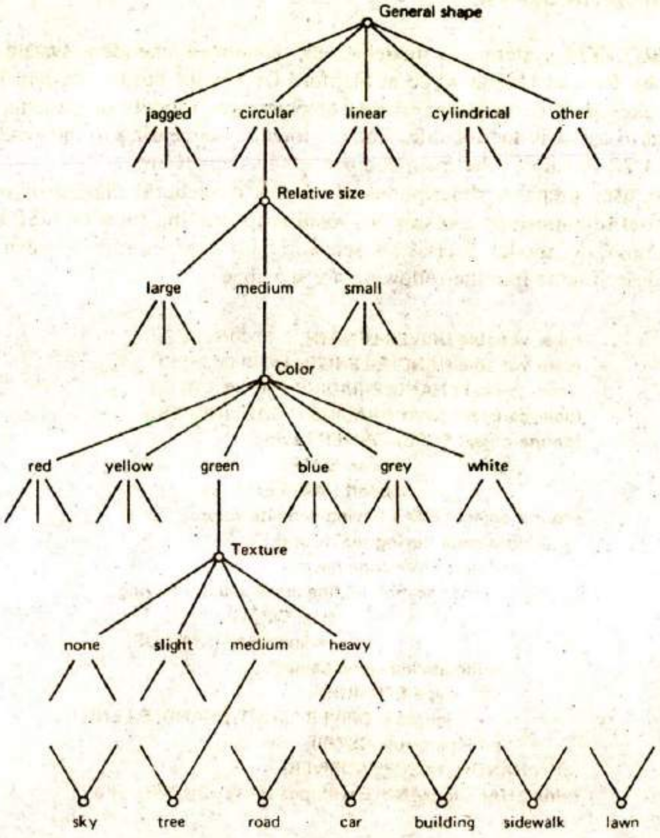


Figure 14.27 Object identification tree.

Objects, with their attributes and relations are then used to construct an associative net scene, a frame network, or other structure.

14.6 VISION SYSTEM ARCHITECTURES

In this section we present two vision systems which are somewhat representative of complete system architectures. The first system is a model-based system, one of the earliest successful vision systems. The second is a color region analyzer recently developed at the University of Kyoto, Japan.

The ACRONYM System

The ACRONYM system is a model-based, domain independent system developed by Rodney Brooks (1981) while at Stanford University during the late 1970s. The system takes user descriptions of sets of objects as models or patterns which are then used to assist in the identification of structures appearing in monocular images. Figure 14.28 illustrates the main components of the system.

The user prepares descriptions of objects or general classes of objects and their spatial relationships and subclass relationships in the form of LISP statements. For example, to model a class of screwdrivers with lengths between 1 and 10 inches, descriptions like the following are specified.

```
(user-variable DRIVER-LENGTH (* 10.0 INCHES))
(user-variable HANDLE-LENGTH (* 4.0 INCHES))
(user-constant HANDLE-RADIUS (* 0.5 INCHES))
(user-constant SHAFT-RADIUS (* 0.125 INCHES))
(define object SCREWDRIVER having
               subpart SHAFT
               subpart HANDLE)
(define object SHAFT having cone-descriptor
   (define cone having main-cone
      (define simple-cone having
          cross-section (define cross-section having
                               type CIRCLE
                               radius SHAFT-RADIUS)
        spine (define spine having
              type STRAIGHT
              length (- DRIVER-LENGTH HANDLE-LENGTH))
        sweeping-rule CSW)))
(affix HANDLE to SCREWDRIVER)
(affix SHAFT to HANDLE with pos HANDLE-LENGTH 0 0)
. . .
```

The user descriptions are parsed and transformed by the system into geometric and algebraic network representations. These representations provide volumetric descriptions in local coordinate systems. A graphic presentation, the system's interpretation of the input models created by the user, provides feedback to the user during the modeling process. The completed representations are used by the system to predict what features (e.g. shape, orientation, and position) of the modeled objects can be observed from the input image components. The predicted models are stored as prediction graphs.

The visual input consists of gray-level image processing arrays, a line finder, and an edge linker. This part of the system provides descriptions of objects as defined by segmented edge structures. The descriptions created from this unit are represented as observation graphs. One output from the predictor serves as an input

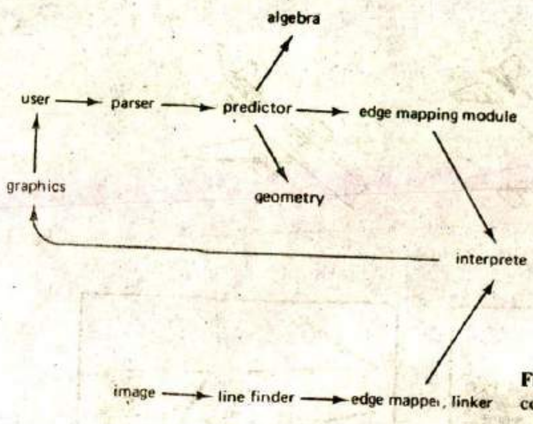


Figure 14.28 Main functional components of ACRONYM.

to the edge mapping and linking module. This unit uses the predicted information (predicted edges, ribbons, or ellipses in the modeled objects) to assist in finding and identifying image objects appearing in the input image. Outputs from both the predictor and the edge mapper and linker serve as inputs to the interpreter. The interpreter is essentially a graph matcher. It tries to find the most matches among subgraphs of the image observation graph and the prediction graph. Each match becomes an interpretation graph. Partial matching is accommodated in the interpretation process through consistency checks.

The basic interpretation process is summarized in Figure 14.29 where models are given for two wide bodied aircraft, (a Boeing 747 and a Lockheed L-1011), and the interpretation of an aircraft from gray-level image to ACRONYM's interpretation is shown.

Ohta's Color Scene Analyzer

Yuichi Ohta of Kyoto University recently developed a vision system which performs region analysis on outdoor color scenes (1986). Outdoor scenes typically include objects such as trees, bushes, sky, roads, buildings, and other objects which are more naturally defined as regions rather than edges. His system makes use of the role color can play in the segmentation process.

Starting with tricolor (red, green, and blue) intensity arrays, digitized images are produced from which regions are defined by a segmentation splitting process. The output of the segmentation process is a two-dimensional array which identifies regions as commonly numbered pixel areas. This array is then transformed into a structured data network which contains descriptive elements for regions such as boundary segments, vertices, and the like. With this network, the system constructs a semantic description of the scene using model knowledge in the form of production rules. Figure 14.30 illustrates the main processing steps carried out by the system.

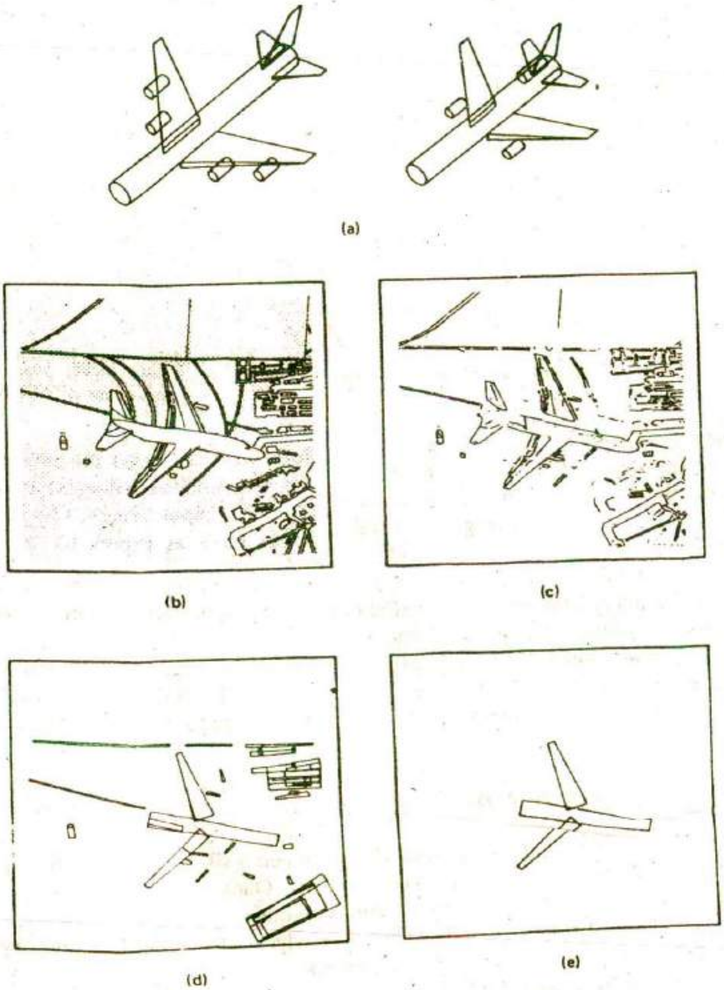


Figure 14.29 Models and stages of interpretation in ACRONYM. (Courtesy of Rodney A. Brooks.)

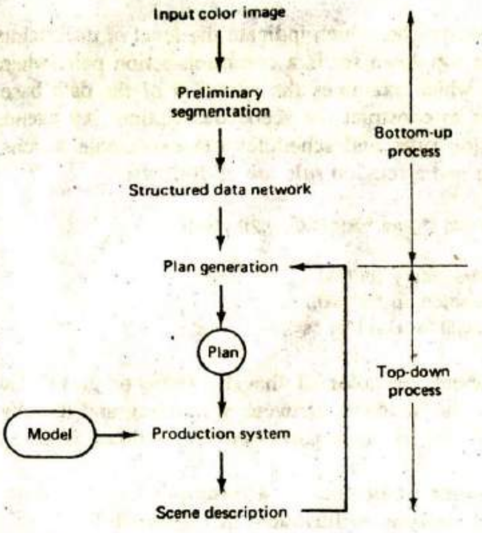


Figure 14.30 Color scene region analyzer.

In the preliminary segmentation stage, the image is segmented into coherent regions using region splitting methods based on color information. Multihistograms serve as cues for thresholding and region splitting. Color features are selected based on results from the Karhunen-Loeve transformation (Devijver and Kittler, 1982) which amounts to choosing those features having the greatest discriminating power (essentially the greatest variance). These segmented regions become the atomic elements from which the structured data network is built.

Regions are characterized by their boundary segments, vertices, line segments, and holes. These basic descriptions are formed during the preliminary segmentation phase. From these, other features are derived including the area, mean color intensities (red, green, blue), degree of texture, contour length, position of mass center, number of holes, minimum bounding rectangle, distance from origin, and orientation. These, and region relationships are described in a data structure known as a patchery data structure. Elements in the data network are essentially matched against model knowledge described in the form of production rules. The rule actions then effectively construct the scene description.

A *plan* is a representation of the crude structure of the input scene given as object labels and their degree of correctness. It is generated by the bottom-up process to provide clues concerning which knowledge can be applied to different parts of the scene.

Knowledge of the task world is represented by sets of production rules. One set is used in the bottom-up process and the other in the top-down process. Each rule in the bottom-up set has a fuzzy predicate which describes properties of relations between objects. The rules also have weights which indicate the level of uncertainty of the knowledge. Each rule in the top-down set is a condition-action pair, where the condition is a fuzzy predicate which examines the situation of the data base. The action part includes operations to construct the scene description. An agenda manages the activation of production rules and schedules the executable actions. Examples of a typical property rule and a relation rule are as follows:

```
[(GEN (or  (*blue *sk) (*gray *sk)) (1.0 . 0.2)) (*sk)]

[GEN (and (linear-boundary *bl *sk)
          (not (position up *bl *sk)))
            (1.0 . 0.5) for sky) (*bl *sk)]
```

The first rule is a property rule about the color of the sky (blue or gray). The second rule is a relation rule about the boundary between a building and the sky. The boundary between the two has a lot of linear parts, and the building is not on the upper side of that boundary.

The final product of the analyzer is, of course, a description of the scene. This is constructed as a hierarchical network as illustrated in Figure 14.31.

Ohta's system has demonstrated that it can deal with fairly complex scenes, including objects with substructures. To validate this claim, a number of outdoor scenes from the Kyoto University campus were analyzed correctly by the system.

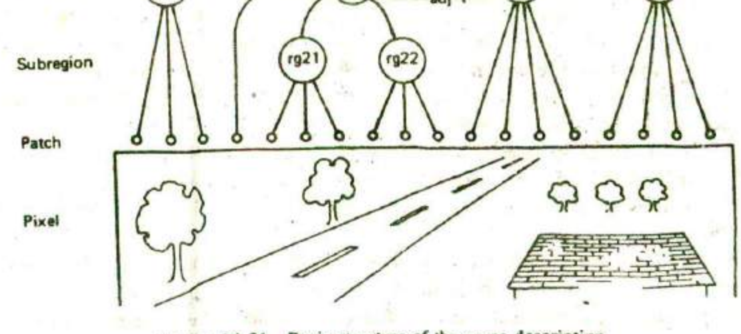


Figure 14.31 Basic structure of the scene description.

14.7 SUMMARY

Computer vision is a computation intensive process. It involves multiple transformations starting with arrays of low level pixels and progressing to high level scene descriptions. The transformation process can be viewed as three stages of processing: low- or early-level processing, intermediate-, and high- or late-level processing. Low-level processing is concerned with the task of finding structure among thousands of atomic gray-level (or tristimulus) pixel intensity values. The goal of this phase is to find and define enough structure in the raw image to permit its segmentation into coherent regions which correspond to definable objects from the source scene. Intermediate-level processing is concerned with the task of accurately forming and describing the segmented regions. The atomic units of this phase are regions and subregions. Finally, high-level processing requires that the segmented regions from the intermediate stage be transformed into scene descriptions. This stage of processing is less mechanical than the two previous stages and relies more on classical AI methods of symbolic processing.

Low-level processing usually involves some form of smoothing operation on the digitized arrays. Smoothing helps to reduce noise and other unwanted features. This is followed by some form of edge detection such as the application of difference operators to the arrays. Edge fragments must then be joined to form continuous contours which outline objects. Various techniques are available for these operations.

The dual-of-the-edge-finding approach is region segmentation, which may be accomplished by region splitting, region growing, or a combination of both. Multihistograms and thresholding are commonly used techniques in the segmentation process. They are applied to one or more image features such as intensity, color, texture, shading, or optical flow in the definition of coherent regions. The end product of the segmentation process will be homogeneous regions. The properties of these regions and their interrelationships must be described in order that they may be identified in the high level processing stage. Regions can be described by boundary segments, vertices, number of holes, compactness, location, orientation, and so on.

The final stage is the knowledge application stage, since the regions must be interpreted and explained. This requires task or domain specific knowledge as well as some general world knowledge. The current state-of-the-art in computer vision does not permit the interpretation of arbitrary complex scenes such as that depicted in Figure 14.1. Much work still remains before that degree of sophistication can be realized.

EXERCISES

14.1. Visual continuity in TV systems is maintained through the generation of 15 frames per second. What characteristics must an ADC unit have to match this same level of continuity for a vision system with a 1024 × 1024 pixel resolution?

14.2. Describe the types of world knowledge a vision system must have to "comprehend" the scene portrayed in Figure 14.3.

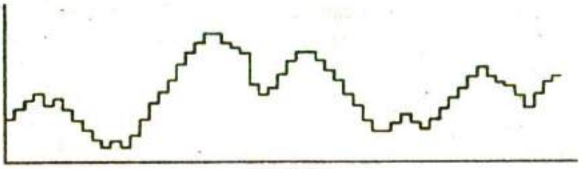
14.3. Suppose the CPU in a vision system takes 200 nanoseconds to perform memory/register transfers and 500 nanoseconds to perform basic arithmetic operations. Estimate the time required to produce a binary image for a system with a resolution of 256 × 256 pixels.

14.4. How much memory is required to produce and compare five different binary images, each with a different threshold level? Assume a system resolution of 512 × 512. Can the binary images be compressed in some way to reduce memory requirements?

14.5. Find the binary image for the array given below when the threshold is set at 35.

$$\begin{bmatrix} 23 & 1 & 32 & 35 \\ 36 & 30 & 42 & 38 \\ 2 & 9 & 34 & 36 \\ 37 & 36 & 35 & 33 \end{bmatrix}$$

14.6. Given the following histogram, what are the most likely threshold points? Explain why you chose the given points and rejected others.



Histogram

14.7. What is the value of the smoothed pixel for the associated mask?

$$\begin{array}{cc} \text{MASK} & \text{PIXELS} \\ \begin{bmatrix} & 3/16 & \\ 3/16 & 1/4 & 3/16 \\ & 3/16 & \end{bmatrix} & \begin{bmatrix} 7 & 8 & 9 \\ 5 & 4 & 6 \\ 4 & 6 & 2 \end{bmatrix} \end{array}$$

14.8. Compare the effects of the eight- and four-neighbor filters described in Section 14.2 when applied to the following array of pixel gray-level values.

$$\begin{bmatrix} 5 & 8 & 8 & 10 & 12 & 29 & 32 & 30 \\ 4 & 7 & 8 & 9 & 10 & 9 & 30 & 29 \\ 5 & 8 & 7 & 8 & 11 & 33 & 31 & 34 \\ 6 & 9 & 8 & 10 & 34 & 31 & 29 & 33 \\ 6 & 8 & 9 & 32 & 30 & 29 & 5 & 6 \\ 8 & 7 & 31 & 32 & 32 & 28 & 6 & 7 \\ 7 & 8 & 33 & 33 & 29 & 7 & 8 & 7 \\ 9 & 30 & 32 & 31 & 28 & 8 & 8 & 9 \end{bmatrix}$$

14.9. Low noise systems should use little or no filtering to avoid unnecessary blurring. This means that more weight should be given to the pixel being smoothed. Define two low-noise filters, one a four-neighbor and one an eight-neighbor filter, and compare their effects on the array of Problem 14.5.

14.10. Using a value of $n = 1$, apply D_x and D_y (horizontally) to the array of Problem 14.5 and comment on the trace of any apparent edges.

14.11. Apply the vector gradient to the array of Problem 14.5 and compare the results to those of Problem 14.7.

14.12. This problem relates to the application of template matching using correlation techniques. The objective is to try to match an unknown two-dimensional curve or waveform with a known waveform. Assume that both waveforms are discrete and are represented as arrays of unsigned numbers. Write a program in any suitable language to match the unknown waveform to the known waveform using the correlation function given as

$$C_i = \frac{<\mathbf{X},\mathbf{Z}_i>}{\|\mathbf{X}\|\ \|\mathbf{Z}_i\|}$$

where $\mathbf{X}$ is the unknown pattern vector, $\mathbf{Z}_i$ is the known pattern vector at position i, $<\mathbf{X},\mathbf{Z}_i>$ denotes the inner product of $\mathbf{X}$ and $\mathbf{Z}_i$, and $\|\mathbf{X}\|$ is the norm of $\mathbf{X}$.

$$\|\mathbf{X}\| = [\Sigma_i x_i^2]^{1/2}$$

14.13 Write a program to apply the Sobel edge detection mask to an array consisting of 256×256 pixel gray level values.

14.14 Color and texture are both potentially useful in defining regions. Describe an algorithm that could be used to determine regions that are homogenious in color.

14.15 Referring to Problem 14.14, develop an algorithm that can be used to define regions that are homogeneous in texture.

14.16 Referring to the two previous problems, develop an algorithm that determines regions on the basis of homogeniety in both color and texture.

15
Expert Systems Architectures

This chapter describes the basic architectures of knowledge-based systems with emphasis placed on expert systems. Expert systems are a recent product of artificial intelligence. They began to emerge as university research systems during the early 1970s. They have now become one of the more important innovations of AI since they have been shown to be successful commercial products as well as interesting research tools.

Expert systems have proven to be effective in a number of problem domains which normally require the kind of intelligence possessed by a human expert. The areas of application are almost endless. Wherever human expertise is needed to solve problems, expert systems are likely candidates for application. Application domains include law, chemistry, biology, engineering, manufacturing, aerospace, military operations, finance, banking, meteorology, geology, geophysics, and more. The list goes on and on.

In this chapter we explore expert system architectures and related building tools. We also look at a few of the more important application areas as well. The material is intended to acquaint the reader with the basic concepts underlying expert systems and to provide enough of the fundamentals needed to build basic systems or pursue further studies and conduct research in the area.

15.1 INTRODUCTION

An expert system is a set of programs that manipulate encoded knowledge to solve problems in a specialized domain that normally requires human expertise. An expert system's knowledge is obtained from expert sources and coded in a form suitable for the system to use in its inference or reasoning processes. The expert knowledge must be obtained from specialists or other sources of expertise, such as texts, journal articles, and data bases. This type of knowledge usually requires much training and experience in some specialized field such as medicine, geology, system configuration, or engineering design. Once a sufficient body of expert knowledge has been acquired, it must be encoded in some form, loaded into a knowledge base, then tested, and refined continually throughout the life of the system.

Characteristic Features of Expert Systems

Expert systems differ from conventional computer systems in several important ways.

1. Expert systems use knowledge rather than data to control the solution process. "In the knowledge lies the power" is a theme repeatedly followed and supported throughout this book. Much of the knowledge used is heuristic in nature rather than algorithmic.

2. The knowledge is encoded and maintained as an entity separate from the control program. As such, it is not compiled together with the control program itself. This permits the incremental addition and modification (refinement) of the knowledge base without recompilation of the control programs. Furthermore, it is possible in some cases to use different knowledge bases with the same control programs to produce different types of expert systems. Such systems are known as expert system shells since they may be loaded with different knowledge bases.

3. Expert systems are capable of explaining how a particular conclusion was reached, and why requested information is needed during a consultation. This is important as it gives the user a chance to assess and understand the system's reasoning ability, thereby improving the user's confidence in the system.

4. Expert systems use symbolic representations for knowledge (rules, networks, or frames) and perform their inference through symbolic computations that closely resemble manipulations of natural language. (An exception to this is the expert system based on neural network architectures.)

5. Expert systems often reason with metaknowledge; that is, they reason with knowledge about themselves, and their own knowledge limits and capabilities.

Background History

Expert systems first emerged from the research laboratories of a few leading U.S. universities during the 1960s and 1970s. They were developed as specialized problem solvers which emphasized the use of knowledge rather than algorithms and general search methods. This approach marked a significant departure from conventional AI systems architectures at the time. The accepted direction of researchers then was to use AI systems that employed general problem solving techniques such as hill-climbing or means-end analysis (Chapter 9) rather than specialized domain knowledge and heuristics. This departure from the norm proved to be a wise choice. It led to the development of a new class of successful systems and special system designs.

The first expert system to be completed was DENDRAL, developed at Stanford University in the late 1960s. This system was capable of determining the structure of chemical compounds given a specification of the compound's constituent elements and mass spectrometry data obtained from samples of the compound. DENDRAL used heuristic knowledge obtained from experienced chemists to help constrain the problem and thereby reduce the search space. During tests, DENDRAL discovered a number of structures previously unknown to expert chemists.

As researchers gained more experience with DENDRAL, they found how difficult it was to elicit expert knowledge from experts. This led to the development of Meta-DENDRAL, a learning component for DENDRAL which was able to learn rules from positive examples, a form of inductive learning described later in detail (Chapters 18 and 19).

Shortly after DENDRAL was completed, the development of MYCIN began at Stanford University. MYCIN is an expert system which diagnoses infectious blood diseases and determines a recommended list of therapies for the patient. As part of the Heuristic Programming Project at Stanford, several projects directly related to MYCIN were also completed including a knowledge acquisition component called THEIRESIUS, a tutorial component called GUIDON, and a shell component called EMYCIN (for Essential MYCIN). EMYCIN was used to build other diagnostic systems including PUFF, a diagnostic expert for pulmonary diseases. EMYCIN also became the design model for several commercial expert system building tools.

MYCIN's performance improved significantly over a several year period as additional knowledge was added. Tests indicate that MYCIN's performance now equals or exceeds that of experienced physicians. The initial MYCIN knowledge base contained about only 200 rules. This number was gradually increased to more than 600 rules by the early 1980s. The added rules significantly improved MYCIN's performance leading to a 65% success record which compared favorably with experienced physicians who demonstrated only an average 60% success rate (Lenat, 1984). (An example of MYCIN's rules is given in Section 4.9, and the treatment of uncertain knowledge by MYCIN is described in Section 6.5.)

Other early expert system projects included PROSPECTOR, a system that assists geologists in the discovery of mineral deposits, and R1 (aka XCON), a system used by the Digital Equipment Corporation to select and configure components of complex computer systems. Since the introduction of these early expert systems, numerous commercial and military versions have been completed with a high degree of success. Some of these application areas are itemized below.

Applications

Since the introduction of these early expert systems, the range and depth of applications has broadened dramatically. Applications can now be found in almost all areas of business and government. They include such areas as

Different types of medical diagnoses (internal medicine, pulmonary diseases, infectious blood diseases, and so on)

Diagnosis of complex electronic and electromechanical systems

Diagnosis of diesel electric locomotion systems

Diagnosis of software development projects

Planning experiments in biology, chemistry, and molecular genetics

Forecasting crop damage

Identification of chemical compound structures and chemical compounds

Location of faults in computer and communications systems

Scheduling of customer orders, job shop production operations, computer resources for operating systems, and various manufacturing tasks

Evaluation of loan applicants for lending institutions

Assessment of geologic structures fro n dip meter logs

Analysis of structural systems for design or as a result of earthquake damage

The optimal configuration of components to meet given specifications for a complex system (like computers or manufacturing facilities)

Estate planning for minimal taxation and other specified goals

Stock and bond portfolio selection and management

The design of very large scale integration (VLSI) systems

Numerous military applications ranging from battlefield assessment to ocean surveillance

Numerous applications related to space planning and exploration

Numerous areas of law including civil case evaluation, product liability, assault and battery, and general assistance in locating different law precedents

Planning curricula for students

Teaching students specialized tasks (like trouble shooting equipment faults)

Importance of Expert Systems

The value of expert systems was well established by the early 1980s. A number of successful applications had been completed by then and they proved to be cost effective. An example which illustrates this point well is the diagnostic system developed by the Campbell Soup Company.

Campbell Soup uses large sterilizers or cookers to cook soups and other canned products at eight plants located throughout the country. Some of the larger cookers hold up to 68,000 cans of food for short periods of cooking time. When difficult maintenance problems occur with the cookers, the fault must be found and corrected quickly or the batch of foods being prepared will spoil. Until recently, the company had been depending on a single expert to diagnose and cure the more difficult problems, flying him to the site when necessary. Since this individual will retire in a few years taking his expertise with him, the company decided to develop an expert system to diagnose these difficult problems.

After some months of development with assistance from Texas Instruments, the company developed an expert system which ran on a PC. The system has about 150 rules in its knowledge base with which to diagnose the more complex cooker problems. The system has also been used to provide training to new maintenance personnel. Cloning multiple copies for each of the eight locations cost the company only a few pennies per copy. Furthermore, the system cannot retire, and its performance can continue to be improved with the addition of more rules. It has already proven to be a real asset to the company. Similar cases now abound in many diverse organizations.

15.2 RULE-BASED SYSTEM ARCHITECTURES

The most common form of architecture used in expert and other types of knowledge-based systems is the production system, also called the rule-based system. This type of system uses knowledge encoded in the form of production rules, that is, if . . . then rules. We may remember from Chapter 4 that rules have an antecedent or condition part, the left-hand side, and a conclusion or action part, the right-hand side.

```
    IF: Condition-1 and Condition-2 and Condition-3
  THEN: Take Action-4

    IF: The temperature is greater than 200 degrees, and
        The water level is low
  THEN: Open the safety valve.
```

$A \& B \& C \& D \rightarrow E \& F$

Each rule represents a small chunk of knowledge relating to the given domain of expertise. A number of related rules collectively may correspond to a chain of inferences which lead from some initially known facts to some useful conclusions. When the known facts support the conditions in the rule's left side, the conclusion or action part of the rule is then accepted as known (or at least known with some degree of certainty). Examples of some typical expert system rules were described in earlier sections (for example see Sections 4.9, 6.5, and 10.6).

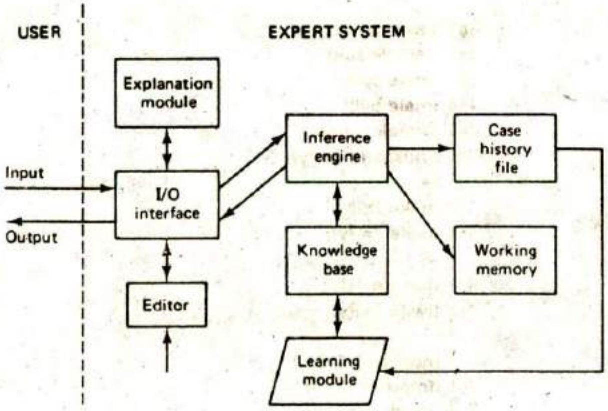


Figure 15.1 Components of a typical expert system.

Inference in production systems is accomplished by a process of chaining through the rules recursively, either in a forward or backward direction, until a conclusion is reached or until failure occurs. The selection of rules used in the chaining process is determined by matching current facts against the domain knowledge or variables in rules and choosing among a candidate set of rules the ones that meet some given criteria, such as specificity. The inference process is typically carried out in an interactive mode with the user providing input parameters needed to complete the rule chaining process.

The main components of a typical expert system are depicted in Figure 15.1. The solid lined boxes in the figure represent components found in most systems whereas the broken lined boxes are found in only a few such systems.

The Knowledge Base

The knowledge base contains facts and rules about some specialized knowledge domain. An example of a simple knowledge base giving family relationships is illustrated in Figure 15.2. The rules in this figure are given in the same LISP format as those of Section 10.6 which is similar to the format given in the OPS5 language as presented by Bronston, Farrell, Kant, and Martin (1985). Each fact and rule is identified with a name (*a*1, *a*2, . . . , *r*1, *r*2, . . .). For ease in reading, the left side is separated from the right by the implication symbol →. Conjuncts on the left are given within single parentheses (sublists), and one or more conclusions may follow the implication symbol. Variables are identified as a symbol preceded by a question mark. It should be noted that rules found in real working systems may have many conjuncts in the LHS. For example, as many as eight or more are not uncommon.

```
((a1 (male bob))
 (a2 (female sue))
 (a3 (male sam))
 (a4 (male bill))
 (a5 (female pam))
 (r1 ((husband ?x ?y))
     →
     (male ?x))
 (r2 ((wife ?x ?y))
     →
     (female ?x))
 (r3 ((wife ?x ?y))
     →
     (husband ?y ?x))
 (r4 ((mother ?x ?y)
      (husband ?z ?x))
     →
     (father ?z ?y))
 (r5 ((father ?x ?y))
      (wife ?z ?x))
     →
     (mother ?z ?y))
 (r6 ((husband ?x ?y))
     →
     (wife ?y ?x))
 (r7 ((father ?x ?z)
      (mother ?y ?z))
     →
     (husband ?x ?y))
 (r8 ((father ?x ?z)
      (mother ?y ?z))
     →
     (wife ?y ?z))
 (r9 ((father ?x ?y)
      (father ?y ?z))
     →
     (grandfather ?x ?z)))
```

Figure 15.2 Facts and rules in a simple knowledge base.

In PROLOG, rules are written naturally as clauses with both a head and body. For example, a rule about a patient's symptoms and the corresponding diagnosis of hepatitis might read in English as the rule

IF: The patient has a chronic disorder, and
the sex of the patient is female, and
the age of the patient is less than 30, and
the patient shows condition A, and
test B reveals biochemistry condition C

THEN: conclude the patient's diagnosis is autoimmune-chronic-hepatitis.

This rule could be written straightaway in PROLOG as

```
conclude(patient, diagnosis, autoimmune_chronic_hepatitis):-
        same(patient, disorder, chronic),
        same(patient, sex, female),
        lessthan(patient, age, 30),
        same(patient, symptom_a, value_a),
        same(patient, biochemistry, value_c).
```

Note that PROLOG rules have at most one conclusion clause.

The Inference Process

The inference engine accepts user input queries and responses to questions through the I/O interface and uses this dynamic information together with the static knowledge (the rules and facts) stored in the knowledge base. The knowledge in the knowledge base is used to derive conclusions about the current case or situation as presented by the user's input.

The inferring process is carried out recursively in three stages: (1) match, (2) select, and (3) execute. During the match stage, the contents of working memory are compared to facts and rules contained in the knowledge base. When consistent matches are found, the corresponding rules are placed in a conflict set. To find an appropriate and consistent match, substitutions (instantiations) may be required. Once all the matched rules have been added to the conflict set during a given cycle, one of the rules is selected for execution. The criteria for selection may be most recent use, rule condition specificity, (the number of conjuncts on the left), or simply the smallest rule number. The selected rule is then executed and the right-hand side or action part of the rule is then carried out. Figure 15.3 illustrates this match-select-execute cycle.

As an example, suppose the working memory contains the two clauses

```
(father bob sam)
(mother sue sam)
```

When the match part of the cycle is attempted, a consistent match will be made between these two clauses and rules r7 and r8 in the knowledge base. The match is made by substituting Bob for ?*x*, Sam for ?*z*, and Sue for ?*y*. Consequently, since all the conditions on the left of both r7 and r8 are satisfied, these two rules will be placed in the conflict set. If there are no other working memory clauses to match, the selection step is executed next. Suppose, for one or more of the selection criteria stated above, r7 is the rule chosen to execute. The clause on the right side of r7 is instantiated and the execution step is initiated. The execution step may result in the right-hand clause (husband bob sue) being placed in working memory or it may be used to trigger a message to the user. Following the execution step, the match-select-execute cycle is repeated.

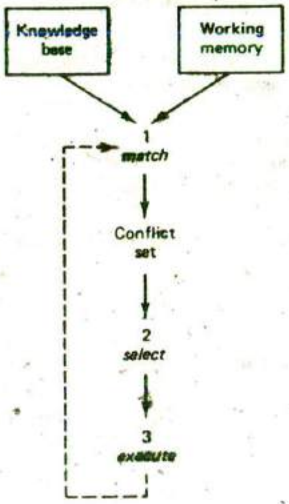


Figure 15.3 The production system inference cycle.

As another example of matching, suppose the two facts (a6 (father sam bill)) and (a7 (father bill pam)) have been added to the knowledge base and the immediate goal is a query about Pam's grandfather. When made, assume this query has resulted in placement of the clause (grandfather ?x pam) into working memory. For this goal to succeed, consistent substitutions must be made for the variables ?*x* and ?*y* in rule r9 with a6 and a7. This will be the case if Sam and Bill are substituted for ?*x* and ?y in the subgoal left-hand conditions of r9. The right hand side will then correctly state that Pam's grandfather is Sam.

When the left side of a sequence of rules is instantiated first and the rules are executed from left to right, the process is called forward chaining. This is also known as data-driven inference since input data are used to guide the direction of the inference process. For example, we can chain forward to show that when a student is encouraged, is healthy, and has goals, the student will succeed.

```
ENCOURAGED(student) → MOTIVATED(student)
MOTIVATED(student) & HEALTHY(student) → WORKHARD(student)
WORKHARD(student) & HASGOALS(student) → EXCELL(student)
EXCELL(student) → SUCCEED(student)
```

On the other hand, when the right side of the rules is instantiated first, the left-hand conditions become subgoals. These subgoals may in turn cause sub-subgoals to be established, and so on until facts are found to match the lowest subgoal conditions. When this form of inference takes place, we say that backward chaining is performed. This form of inference is also known as goal-driven inference since an initial goal establishes the backward direction of the inferring.

For example, in MYCIN the initial goal in a consultation is "Does the patient have a certain disease?" This causes subgoals to be established such as "are certain bacteria present in the patient?" Determining if certain bacteria are present may require such things as tests on cultures taken from the patient. This process of setting up subgoals to confirm a goal continues until all the subgoals are eventually satisfied or fail. If satisfied, the backward chain is established thereby confirming the main goal.

When rules are executed, the resulting action may be the placement of some new facts in working memory, a request for additional information from the user, or simply the stopping of the search process. If the appropriate knowledge has been stored in the knowledge base and all required parameter values have been provided by the user, conclusions will be found and will be reported to the user. The chaining continues as long as new matches can be found between clauses in the working memory and rules in the knowledge base. The process stops when no new rules can be placed in the conflict set.

Some systems use both forward and backward chaining, depending on the type of problem and the information available. Likewise, rules may be tested exhaustively or selectively, depending on the control structure. In MYCIN, rules in the KB are tested exhaustively. However, when the number of rules exceeds a few hundred, this can result in an intolerable amount of searching and matching. In such cases, techniques such as those found in the RETE algorithm (Chapter 10) may be used to limit the search.

Many expert systems must deal with uncertain information. This will be the case when the evidence supporting a conclusion is vague, incomplete, or otherwise uncertain. To accommodate uncertainties, some form of probabilities, certainty factors, fuzzy logic, heuristics, or other methods must be introduced into the inference process. These methods were introduced in Chapters 5 and 6. The reader is urged at this time to review those methods to see how they may be applied to expert systems.

Explaining How or Why

The explanation module provides the user with an explanation of the reasoning process when requested. This is done in response to a how query or a why query.

To respond to a how query, the explanation module traces the chain of rules fired during a consultation with the user. The sequence of rules that led to the conclusion is then printed for the user in an easy to understand human-language style. This permits the user to actually see the reasoning process followed by the system in arriving at the conclusion. If the user does not agree with the reasoning steps presented, they may be changed using the editor.

To respond to a why query, the explanation module must be able to explain why certain information is needed by the inference engine to complete a step in the reasoning process before it can proceed. For example, in diagnosing a car that will not start, a system might be asked why it needs to know the status of the distributor spark. In response, the system would reply that it needs this information to determine if the problem can be isolated to the ignition system. Again, this information allows the user to determine if the system's reasoning steps appear to be sound. The explanation module programs give the user the important ability to follow the inferencing steps at any time during the consultation.

Building a Knowledge Base

The editor is used by developers to create new rules for addition to the knowledge base, to delete outmoded rules, or to modify existing rules in some way. Some of the more sophisticated expert system editors provide the user with features not found in typical text editors, such as the ability to perform some types of consistency tests for newly created rules, to add missing conditions to a rule, or to reformat a newly created rule. Such systems also prompt the user for missing information, and provide other general guidance in the KB creation process.

One of the most difficult tasks in creating and maintaining production systems is the building and maintaining of a consistent but complete set of rules. This should be done without adding redundant or unnecessary rules. Building a knowledge base requires careful planning, accounting, and organization of the knowledge structures. It also requires thorough validation and verification of the completed knowledge base, operations which have yet to be perfected. An "intelligent" editor can greatly simplify the process of building a knowledge base.

TEIRESIAS (Davis, 1982) is an example of an intelligent editor developed to assist users in building a knowledge base directly without the need for an intermediary knowledge engineer. TEIRESIUS was developed to work with systems like MYCIN in providing a direct user-to-system dialog. TEIRESIUS assists the user in formulating, checking, and modifying rules for inclusion in the performance program's knowledge base. For this, TEIRESIUS uses some metaknowledge, that is, knowledge about MYCIN's knowledge. The dialog is carried out in a near English form so that the user needs to know little about the internal form of the rules.

The I/O Interface

The input-output interface permits the user to communicate with the system in a more natural way by permitting the use of simple selection menus or the use of a restricted language which is close to a natural language. This means that the system must have special prompts or a specialized vocabulary which encompasses the terminology of the given domain of expertise. For example, MYCIN can recognize many medical terms in addition to various common words needed to communicate. For this, MYCIN has a vocabulary of some 2000 words.

Personal Consultant Plus, a commercial PC version of the MYCIN architecture, uses menus and English prompts to communicate with the user. The prompts, written in standard English, are provided by the developer during the system building stage. How and why explanations are also given in natural language form.

The learning module and history file are not common components of expert systems. When they are provided, they are used to assist in building and refining the knowledge base. Since learning is treated in great detail in later chapters, no description is given here.

15.3 NONPRODUCTION SYSTEM ARCHITECTURES

Other, less common expert system architectures (although no less important) are those based on nonproduction rule-representation schemes. Instead of rules, these systems employ more structured representation schemes like associative or semantic networks, frame and rule structures, decision trees, or even specialized networks like neural networks. In this section we examine some typical system architectures based on these methods.

Associative or Semantic Network Architectures

Associative or semantic network representation schemes were discussed in Chapter 7. From the description there, we know that an associative network is a network made up of nodes connected by directed arcs. The nodes represent objects, attributes, concepts, or other basic entities, and the arcs, which are labeled, describe the relationship between the two nodes they connect. Special network links include the ISA and HASPART links which designate an object as being a certain type of object (belonging to a class of objects) and as being a subpart of another object, respectively.

Associative network representations are especially useful in depicting hierarchical knowledge structures, where property inheritance is common. Objects belonging to a class of other objects may inherit many of the characteristics of the class. Inheritance can also be treated as a form of default reasoning. This facilitates the storage of information when shared by many objects as well as the inferencing process.

Associative network representations are not a popular form of representation for standard expert systems. More often, these network representations are used in natural language or computer vision systems or in conjunction with some other form of representation.

One expert system based on the use of an associative network representation is CASNET (Causal Associational Network), which was developed at Rutgers University during the early 1970s (Weiss et al., 1978). CASNET is used to diagnose and recommend treatment for glaucoma, one of the leading causes of blindness.

The network in CASNET is divided into three planes or types of knowledge as depicted in Figure 15.4. The different knowledge types are

Patient observations (tests, symptoms, other signs)
Pathophysiological states
Disease categories

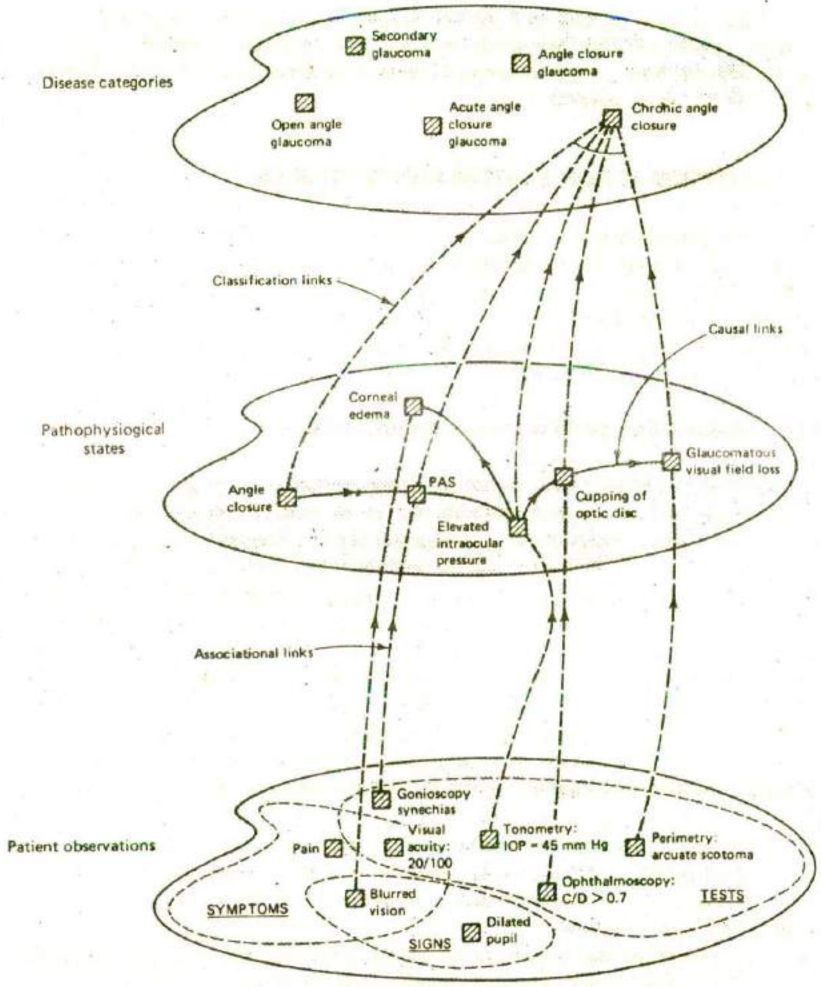


Figure 15.4 Levels of network description in CASNET. (From Artificial Intelligence Journal, Vol. II p. 148, 1978. By permission.)

Patient observations are provided by the user during an interactive session with the system. The system presents menu type queries, and the user selects one of several possible choices. These observations help to establish an abnormal condition caused by a disease process. The condition is established through the causal network model as part of the cause and effect relationship relating symptoms and other signs to diseases.

Inference is accomplished by traversing the network, following the most plausible paths of causes and effects. Once a sufficiently strong path has been determined through the network, diagnostic conclusions are inferred using classification tables that interpret patterns of the causal network. These tables are similar to rule interpretations.

The CASNET system was never used much beyond the initial research stage. At the time, physicians were reluctant to use computer systems in spite of performance tests in which CASNET scored well.

Frame Architectures

Frame representations were described in Chapter 7. Frames are structured sets of closely related knowledge, such as an object or concept name, the object's main attributes and their corresponding values, and possibly some attached procedures (if-needed, if-added, if-removed procedures). The attributes, values, and procedures are stored in specified slots and slot facets of the frame. Individual frames are usually linked together as a network much like the nodes in an associative network. Thus, frames may have many of the features of associative networks, namely, property inheritance and default reasoning. Several expert systems have been constructed with frame architectures, and a number of building tools which create and manipulate frame structured systems have been developed.

An example of a frame-based system is the PIP system (Present Illness Program) developed at M.I.T. during the late 1970s and 1980s (Szolovits and Pauker, 1978). This system was used to diagnose patients using low cost, easily obtained information, the type of information obtained by a general practitioner during an office examination.

The medical knowledge in PIP is organized in frame structures, where each frame is composed of categories of slots with names such as

Typical findings
Logical decision criteria
Complimentary relations to other frames
Differential diagnosis
Scoring

The patient findings are matched against frames, and when a close match is found, a trigger status occurs. A trigger is a finding that is so strongly related to a disorder that the system regards it as an active hypothesis, one to be pursued further. A special is-sufficient slot is used to confirm the presence of a disease when key findings correlate with the slot contents.

Decision Tree Architectures

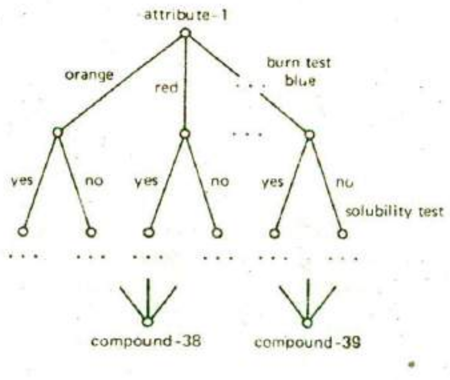
Knowledge for expert systems may be stored in the form of a decision tree when the knowledge can be structured in a top-to-bottom manner. For example, the identification of objects (equipment faults, physical objects, diseases, and the like) can be made through a decision tree structure. Initial and intermediate nodes in the tree correspond to object attributes, and terminal nodes correspond to the identities of objects. Attribute values for an object determine a path to a leaf node in the tree which contains the object's identification. Each object attribute corresponds to a nonterminal node in the tree and each branch of the decision tree corresponds to an attribute value or set of values.

A segment of a decision tree knowledge structure taken from an expert system used to identify objects such as liquid chemical waste products is illustrated in Figure 15.5 (Patterson, 1987). Each node in the tree corresponds to an identifying attribute such as molecular weight, boiling point, burn test color, or solubility test results. Each branch emanating from a node corresponds to a value or range of values for the attribute such as 20–37 degrees C, yellow, or nonsoluble in sulphuric acid.

An identification is made by traversing a path through the tree (or network) until the path leads to a unique leaf node which corresponds to the unknown object's identity.

The knowledge base, which is the decision tree for an identification system, can be constructed with a special tree-building editor or with a learning module. In either case, a set of the most discriminating attributes for the class of objects being identified should be selected. Only those attributes that discriminate well among different objects need be used. Permissible values for each of the attributes are grouped into separable sets, and each such set determines a branch from the attribute node to the next node.

New nodes and branches can be added to the tree when additional attributes



15.5 A segment of a decision tree structure.

are needed to further discriminate among new objects. As the system gains experience, the values associated with the branches can be modified for more accurate results.

Blackboard System Architectures

Blackboard architectures refer to a special type of knowledge-based system which uses a form of opportunistic reasoning. This differs from pure forward or pure backward chaining in production systems in that either direction may be chosen dynamically at each stage in the problem solution process. Other reasoning methods (model driven, for example) may also be used.

Blackboard systems are composed of three functional components as depicted in Figure 15.6.

1. There are a number of *knowledge sources* which are separate and independent sets of coded knowledge. Each knowledge source may be thought of as a specialist in some limited area needed to solve a given subset of problems. The sources may contain knowledge in the form of procedures, rules, or other schemes.
2. A globally accessible data base structure, called a *blackboard*, contains the current problem state and information needed by the knowledge sources (input data, partial solutions, control data, alternatives, final solutions). The knowledge sources make changes to the blackboard data that incrementally lead to a solution. Communication and interaction between the knowledge sources takes place solely through the blackboard.
3. *Control information* may be contained within the sources, on the blackboard, or possibly in a separate module. (There is no actual control unit specified as

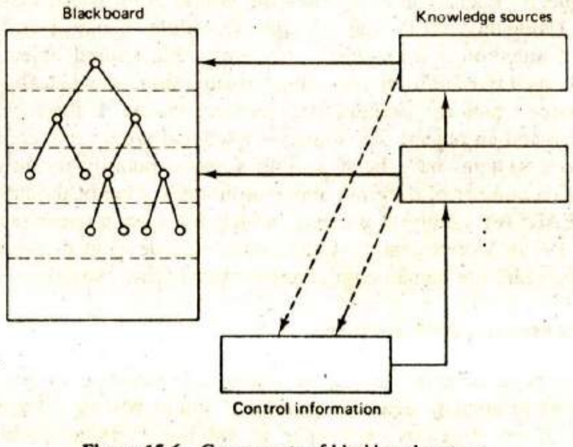


Figure 15.6 Components of blackboard systems.

part of a blackboard system.) The control knowledge monitors the changes to the blackboad and determines what the immediate focus of attention should be in solving the problem.

H. Penny Nii (1986a) has aptly described the blackboard problem solving strategy through the following analogy.

> Imagine a room with a large blackboard on which a group of experts are piecing together a jigsaw puzzle. Each of the experts has some special knowledge about solving puzzles (e.g., a border expert, a shape expert, a color expert, etc.). Each member examines his or her pieces and decides if they will fit into the partially completed puzzle. Those members having appropriate pieces go up to the blackboard and update the evolving solution. The whole puzzle can be solved in complete silence with no direct communication among members of the group. Each person is self-activating, knowing when he or she can contribute to the solution. The solution evolves in this incremental way with each expert contributing dynamically on an opportunistic basis, that is, as the opportunity to contribute to the solution arises.
>
> The objects on the blackboard are hierarchically organized into levels which facilitate analysis and solution. Information from one level serves as input to a set of knowledge sources. The sources modify the knowledge and place it on the same or different levels.

The control information is used by the control module to determine the focus of attention. This determines the next item to be processed. The focus of attention can be the choice of knowledge sources or the blackboard objects or both. If both, the control determines which sources to apply to which objects.

Problem solving proceeds with a knowledge source making changes to the blackboard objects. Each source indicates the contribution it can make to the new solution state. Using this information, the control module chooses a focus of attention. If the focus of attention is a knowledge source, a blackboard object is chosen as the context of its invocation. If the focus of attention is a blackboard object, a knowledge source which can process that object is chosen. If the focus of attention is both a source and an object, that source is executed within that context.

Blackboard systems have been gaining some popularity recently. They have been applied to a number of different application areas. One of the first applications was in the HEARSAY family of projects, which are speech understanding systems (Reddy et al., 1976). More recently, systems have been developed to analyze complex scenes, and to model the human cognitive processes (Nii, 1986b).

Analogical Reasoning Architectures

Little work has been done in the area of analogical reasoning systems. Yet this is one of the most promising areas for general problem solving. We humans make extensive use of our previous experience in solving everyday problems. This is because new problems are frequently similar to previously encountered problems.

Expert systems based on analogical architectures solve new problems like humans, by finding a similar problem solution that is known and applying the known solution to the new problem, possibly with some modifications. For example, if we know a method of proving that the product of two even integers is even, we can successfully prove that the product of two odd integers is odd through much the same proof steps. Only a slight modification will be required when collecting product terms in the result. Expert systems using analogical architectures will require a large knowledge base having numerous problem solutions and other previously-encountered situations or episodes. Each such situation should be stored as a unit in memory and be content-indexed for rapid retrieval. The inference mechanism must be able to extend known situations or solutions to fit the current problem and verify that the extended solution is reasonable. The author and one of his students has built a small toy analogical expert system in LISP to demonstrate many of the features needed for such systems (Patterson and Chu, 1988).

Neural Network Architectures

Neural networks are large networks of simple processing elements or nodes which process information dynamically in response to external inputs. The nodes are simplified models of neurons. The knowledge in a neural network is distributed throughout the network in the form of internode connections and weighted links which form the inputs to the nodes. The link weights serve to enhance or inhibit the input stimuli values which are then added together at the nodes. If the sum of all the inputs to a node exceeds some threshold value T, the node executes and produces an output which is passed on to other nodes or is used to produce some output response. In the simplest case, no output is produced if the total input is less than T. In more complex models, the output will depend on a nonlinear activation function.

Neural networks were originally inspired as being models of the human nervous system. They are greatly simplified models to be sure (neurons are known to be fairly complex processors). Even so, they have been shown to exhibit many "intelligent" abilities, such as learning, generalization, and abstraction.

A single node is illustrated in Figure 15.7. The inputs to the node are the values $x_1, x_2, \ldots, x_n$, which typically take on values of -1, 0, 1, or real values within the range $(-1,1)$. The weights $w_1, w_2, \ldots, w_n$, correspond to the synaptic strengths of a neuron. They serve to increase or decrease the effects of the corresponding x_i input values. The sum of the products $x_i \cdot w_i$, $i = 1, 2, \ldots, n$, serve as the total combined input to the node. If this sum is large enough to exceed the

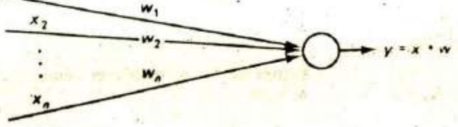


Figure 15.7 Model of a single neuron (node).

threshold amount T, the node fires, and produces an output y, an activation function value placed on the node's output links. This output may then be the input to other nodes or the final output response from the network.

Figure 15.8 illustrates three layers of a number of interconnected nodes. The first layer serves as the input layer, receiving inputs from some set of stimuli. The second layer (called the hidden layer) receives inputs from the first layer and produces a pattern of inputs to the third layer, the output layer. The pattern of outputs from the final layer are the network's responses to the input stimuli patterns. Input links to layer j ($j = 1, 2, 3$) have weights w_{ij} for $i = 1, 2, \ldots, n$.

General multilayer networks having n nodes (number of rows) in each of m layers (number of columns of nodes) will have weights represented as an $n \times m$ matrix **W**. Using this representation, nodes having no interconnecting links will have a weight value of zero. Networks consisting of more than three layers would, of course, be correspondingly more complex than the network depicted in Figure 15.8.

A neural network can be thought of as a black box that transforms the input vector **x** to the output vector **y** where the transformation performed is the result of the pattern of connections and weights, that is, according to the values of the weight matrix **W**.

Consider the vector product

$$\mathbf{x} * \mathbf{w} = \Sigma x_i w_i$$

There is a geometric interpretation for this product. It is equivalent to projecting one vector onto the other vector in n-dimensional space. This notion is depicted in Figure 15.9 for the two-dimensional case.

The magnitude of the resultant vector is given by

$$\mathbf{x} * \mathbf{w} = |\mathbf{x}||\mathbf{w}| \cos \theta$$

where $|\mathbf{x}|$ denotes the norm or length of the vector **x**. Note that this product is maximum when both vectors point in the same direction, that is, when $\theta = 0$. The

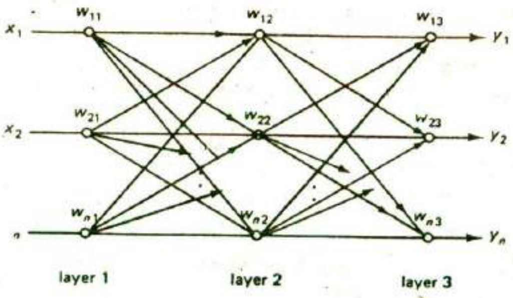


Figure 15.8 A multilayer neural network.

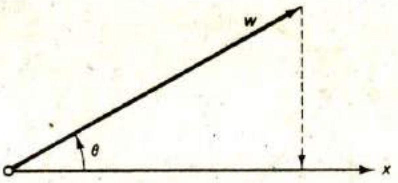


Figure 15.9 Vector multiplication is like vector projection.

product is a minimum when both point in opposite directions or when $\theta = 180$ degrees. This illustrates how the vectors in the weight matrix **W** influence the inputs to the nodes in a neural network.

Learning pattern weights. The interconnections and weights **W** in the neural network store the knowledge possessed by the network. These weights must be preset or learned in some manner. When learning is used, the process may be either supervised or unsupervised. In the supervised case, learning is performed by repeatedly presenting the network with an input pattern and a desired output response. The training examples then consist of the vector pairs $(\mathbf{x},\mathbf{y}')$, where $\mathbf{x}$ is the input pattern and $\mathbf{y}'$ is the desired output response pattern. The weights are then adjusted until the difference between the actual output response $\mathbf{y}$ and the desired response $\mathbf{y}'$ are the same, that is until $D = \mathbf{y} - \mathbf{y}'$ is near zero.

One of the simpler supervised learning algorithms uses the following formula to adjust the weights **W**.

$$\mathbf{W}_{\text{new}} = \mathbf{W}_{\text{old}} + a * D * \frac{\mathbf{x}}{|\mathbf{x}|^2}$$

where $0 < a < 1$ is a learning constant that determines the rate of learning. When the difference D is large, the adjustment to the weights **W** is large, but when the output response $\mathbf{y}$ is close to the target response $\mathbf{y}'$ the adjustment will be small. When the difference D is near zero, the training process terminates at which point the network will produce the correct response for the given input patterns $\mathbf{x}$.

In unsupervised learning, the training examples consist of the input vectors $\mathbf{x}$ only. No desired response $\mathbf{y}'$ is available to guide the system. Instead, the learning process must find the weights w_{ij} with no knowledge of the desired output response. We leave the unsupervised learning description until Chapter 18 where learning is covered in somewhat more detail.

A neural net expert system. An example of a simple neural network diagnostic expert system has been described by Stephen Gallant (1988). This system diagnoses and recommends treatments for acute sarcophagal disease. The system is illustrated in Figure 15.10. From six symptom variables $u_1, u_2, \ldots, u_6$, one of two possible diseases can be diagnosed, u_7 or u_8. From the resultant diagnosis, one of three treatments, u_9, u_{10}, or u_{11} can then be recommended.

When a given symptom is present, the corresponding variable is given a value

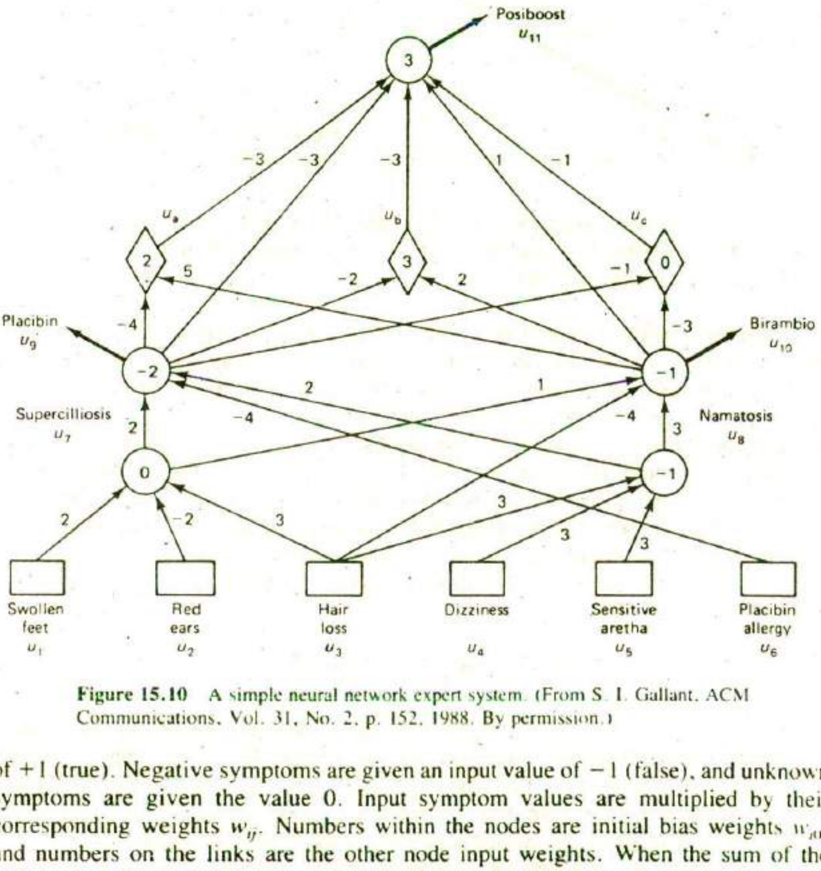


Figure 15.10 A simple neural network expert system. (From S. I. Gallant, ACM Communications, Vol. 31, No. 2, p. 152, 1988. By permission.)

of +1 (true). Negative symptoms are given an input value of −1 (false), and unknown symptoms are given the value 0. Input symptom values are multiplied by their corresponding weights w_{ij}. Numbers within the nodes are initial bias weights w_{i0}, and numbers on the links are the other node input weights. When the sum of the weighted products of the inputs exceeds 0, an output will be present on the corresponding node output and serve as an input to the next layer of nodes.

As an example, suppose the patient has swollen feet ($u_1 = +1$) but not red ears ($u_2 = -1$) nor hair loss ($u_3 = -1$). This gives a value of $u_7 = +1$ (since $0+(2)(1)+(-2)(-1)+(3)(-1) = 1$), suggesting the patient has superciliosis.

When it is also known that the other symptoms of the patient are false ($u_4 = u_5 = u_6 = -1$), it may be concluded that namatosis is absent ($u_8 = -1$), and therefore that birambio ($u_{10} = +1$) should be prescribed while placibin should not be prescribed ($u_9 = -1$). In addition, it will be found that posiboost should also be prescribed ($u_{11} = +1$).

The intermediate triangular shaped nodes were added by the training algorithm. These additional nodes are needed so that weight assignments can be made which permit the computations to work correctly for all training instances.

Deductions can be made just as well when only partial information is available. For example, when a patient has swollen feet and suffers from hair loss, it may be concluded the patient has superciliosis, regardless of whether or not the patient has red ears. This is so because the unknown variable cannot force the sum to change to negative.

A system such as this can also explain how or why a conclusion was reached. For example, when inputs and outputs are regarded as rules, an output can be explained as the conclusion to a rule. If placibin is true, the system might explain why with a statement such as

```
Placibin is TRUE due to the following rule:

          IF Placibin Alergy (u6) is FALSE, and
             Superciliosis is TRUE

          THEN Conclude Placibin is TRUE.
```

Expert systems based on neural network architectures can be designed to possess many of the features of other expert system types, including explanations for how and why, and confidence estimation for variable deduction.

15.4 DEALING WITH UNCERTAINTY

In Chapters 5 and 6 we discussed tne problem of dealing with uncertainty in knowledge-based systems. We found that different approaches were possible including probabilistic (Bayesian or Shafer-Dempster), the use of fuzzy logic, ad-hoc, and heuristic methods. All of these methods have been applied in some form of system, and many building tools permit the use of more than a single method. The ad-hoc method used in MYCIN was described in Section 6.5. Refer to that section to review how uncertainty computations are performed in a typical expert system.

15.5 KNOWLEDGE ACQUISITION AND VALIDATION

One of the most difficult tasks in building knowledge-based systems is in the acquisition and encoding of the requisite domain knowledge. Knowledge for expert systems must be derived from expert sources like experts in the given field, journal articles, texts, reports, data bases, and so on. Elicitation of the right knowledge can take several man years and cost hundreds of thousands of dollars. This process is now recognized as one of the main bottlenecks in building expert and other knowledge-based systems. Consequently, much effort has been devoted to more effective methods of acquisition and coding.

Pulling together and correctly interpreting the right knowledge to solve a set of complex tasks is an onerous job. Typically, experts do not know what specific knowledge is being applied nor just how it is applied in the solution of a given problem. Even if they do know, it is likely they are unable to articulate the problem solving process well enough to capture the low-level knowledge used and the inferring processes applied. This difficulty has led to the use of AI experts (called knowledge engineers) who serve as intermediaries between the domain expert and the system. The knowledge engineer elicits information from the experts and codes this knowledge into a form suitable for use in the expert system.

The knowledge elicitation process is depicted in Figure 15.11. To elicit the requisite knowledge, a knowlege engineer conducts extensive interviews with domain experts. During the interviews, the expert is asked to solve typical problems in the domain of interest and to explain his or her solutions.

Using the knowledge gained from the experts and other sources, the knowledge engineer codes the knowledge in the form of rules or some other representation scheme. This knowledge is then used to solve sample problems for review and validation by the experts. Errors and omissions are uncovered and corrected, and additional knowledge is added as needed. The process is repeated until a sufficient body of knowledge has been collected to solve a large class of problems in the chosen domain. The whole process may take as many as tens of person years.

Penny Nii, an experienced knowledge engineer at Stanford University, has described some useful practices to follow in solving acquisition problems through a sequence of heuristics she uses. They have been summarized in the book *The Fifth Generation* by Feigenbaum and McCorduck (1983) as follows.

> You can't be your own expert. By examining the process of your own expertise you risk becoming like the centipede who got tangled up in her own legs and stopped dead when she tried to figure out how she moved a hundred legs in harmony.
>
> From the beginning, the knowledge engineer must count on throwing efforts away. Writers make drafts, painters make preliminary sketches; knowledge engineers are no different.
>
> The problem must be well chosen. AI is a young field and isn't ready to take on every problem the world has to offer. Expert systems work best when the problem is well bounded, which is computer talk to describe a problem for which large amounts of specialized knowledge may be needed, but not a general knowledge of the world.
>
> If you want to do any serious application you need to meet the expert more than half way; if he's had no exposure to computing, your job will be that much harder.
>
> If none of the tools you normally use works, build a new one.
>
> Dealing with anything but facts implies uncertainty. Heuristic knowledge is not hard and fast and cannot be treated as factual. A weighting procedure has to be built into

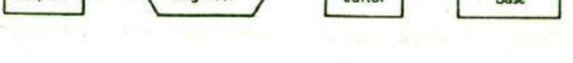


Figure 15.11 The knowledge acquisition process.

the expert system to allow for expressions such as "I strongly believe that . . ." or "The evidence suggests that. . . ."

A high-performance program, or a program that will eventually be taken over by the expert for his own use, must have very easy ways of allowing the knowledge to be modified so that new information can be added and out-of-date information deleted.

The problem needs to be a useful, interesting one. There are knowledge-based programs to solve arcane puzzles, but who cares? More important, the user has to understand the system's real value to his work.

When Nii begins a project, she first persuades a human expert to commit the considerable time that is required to have the expert's mind mined. Once this is done, she immerses herself in the given field, reading texts, articles, and other material to better understand the field and to learn the basic jargon used. She then begins the interviewing process. She asks the expert to describe his or her tasks and problem solving techniques. She asks the expert to choose a moderately difficult problem to solve as an example of the basic approach. This information is then collected, studied, and presented to other members of the development team so that a quick prototype can be constructed for the expert to review. This serves several purposes. First, it helps to keep the expert in the development loop and interested. Secondly, it serves as a rudimentary model with which to uncover flaws and other problems. It also helps both expert and developer in discovering the real way the expert solves problems. This usually leads to a repeat of the problem solving exercise, but this time in a step-by-step walk through of the sample problem. Nii tests the accuracy of the expert's explanations by observing his or her behavior and reliance on data and other sources of information. She is concerned more with the manipulation of the knowledge than with the actual facts. Keeping the expert focused on the immediate problem requires continual prompting and encouragement.

During the whole interview process Nii is mentally examining alternative approaches for the best knowledge representation and inferencing methods to see how well each would best match the expert's behavior. The whole process of elicitation, coding, and verification may take several iterations over a period of several months.

Recognizing the acquisition bottleneck in building expert systems, researchers and vendors alike have sought new and better ways to reduce the burden and reliance placed on knowledge engineers, and in general, ways to improve and speed up the development process. This has led to a number of sophisticated building tools which we consider next.

15.6 KNOWLEDGE SYSTEM BUILDING TOOLS

Since the introduction of the first successful expert systems in the late 1970s, a large number of building tools have been introduced, both by the academic community and industry. These tools range from high level programming languages to intelligent editors to complete shell environment systems. A number of commercial products are now available ranging in price from a few hundred dollars to tens of thousands of dollars. Some are capable of running on medium size PCs while others require larger systems such as LISP machines, minis, or even main frames.

When evaluating building tools for expert system development, the developer should consider the following features and capabilities that may be offered in systems.

1. Knowledge representation methods available (rules, logic-based network structures, frames, with or without inheritance, multiple world views, object-oriented, procedural, and the methods offered for dealing with uncertainty, if any).
2. Inference and control methods available (backward chaining, forward chaining, mixed forward and backward chaining, blackboard architecture approach, logic driven theorem prover, object-oriented approach, the use of attached procedures, types of meta-control capabilities, forms of uncertainty management, hypothetical reasoning, truth maintenance management, pattern matching with or without indexing, user functions permitted, linking options to modules written in other languages, and linking options to other sources of information such as data bases).
3. User interface characteristics (editor flexibility and ease of use, use of menus, use of pop-up windows, developer provided text capabilities for prompts and help messages, graphics capabilities, consistency checking for newly entered knowledge, explanation of how and why capabilities, system help facilities, screen formatting and color selection capabilities, network representation of knowledge base, and forms of compilation available, batch or interactive).
4. General system characteristics and support available (types of applications with which the system has been successfully used, the base programming language in which the system was written, the types of hardware the systems are supported on, general utilities available, debugging facilities, interfacing flexibility to other languages and databases, vendor training availability and cost, strength of software support, and company reputation).

In the remainder of this section, we describe a few representative building tool systems. For a more complete picture of available systems, the reader is referred to other sources.

Personal Consultant Plus

A family of Personal Consultant expert system shells was developed by Texas Instruments, Inc. (TI) in the early 1980s. These shells are rule-based building tools patterned after the MYCIN system architecture and developed to run on a PC as well as on larger systems such as the TI Explorer. The largest and most versatile of the Personal Consultant family is Personal Consultant Plus.

Personal Consultant Plus permits the use of structures called frames (different from the frames of Chapter 7) to organize functionally related production rules into subproblem groups. The frames are hierarchically linked into a tree structure which is traversed during a user consultation. For example, a home electrical appliance diagnostic system might have subframes of microwave cooker, iron, blender, and toaster as depicted in Figure 15.12.

When diagnosing a problem, only the relevant frames would be matched, and the rules in each frame would address only that part of the overall problem. A feature of the frame structure is that parameter property inheritance is supported from ancestor to descendant frames.

The system supports certainty factors both for parameter values and for complete rules. These factors are propagated throughout a chain of rules used during a consultation session, and the resultant certainty values associated with a conclusion are presented. The system also has an explanation capability to show how a conclusion was reached by displaying all rules that lead to a conclusion and why a fact is needed to fire a given rule.

An interactive dialog is carried out between user and system during a consultation session. The system prompts [illegible] with English statements provided by the developer. Menu windows with selectable colors provide the user parameter value selections. A help facility is also available so the developer can provide explanations and give general assistance to the user during a consultation session.

A system editor provides a number of helpful facilities to build, store, and print a knowledge base. Parameter and property values, textual prompts, help messages, and debug traces are all provided through the editor. In addition, user defined functions written in LISP may be provided by a developer as part of a rule's conditions and/or actions. The system also provides access to other sources of information, such as dBase II and dBase III. An optional graphics package is also available at extra cost.

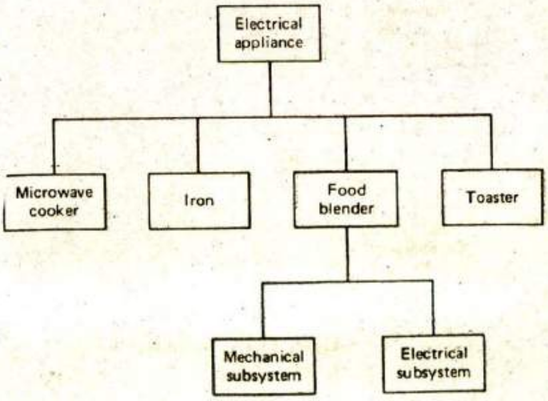


Figure 15.12 Hierarchical frame structure in PC Plus.

Radian Rulemaster

The Rulemaster system developed in the early 1980s by the Radian Corporation was written in C language to run on a variety of mini- and microcomputer systems. Rulemaster is a rule-based building tool which consists of two main components: Radial, a procedural, block structured language for expressing decision rules related to a finite state machine, and Rulemaker, a knowledge acquisition system which induces decision trees from examples supplied by an expert. A program in Rulemaster consists of a collection of related modules which interact to affect changes of state. The modules may contain executable procedures, advice, or data. The building system is illustrated in Figure 15.13.

Rulemaster's knowledge can be based on partial certainty using fuzzy logic or heuristic methods defined by the developer. Users can define their own data types or abstract types much the same as in Pascal. An explanation facility is provided to explain its chain of reasoning. Programs in other languages can also be called from Rulemaster.

One of the unique features of Rulemaster is the Rulemaker component which has the ability to induce rules from examples. Experts are known to have difficulty in directly expressing rules related to their decision processes. On the other hand, they can usually come up with a wealth of examples in which they describe typical solution steps. The examples provided by the expert offer a more accurate way in

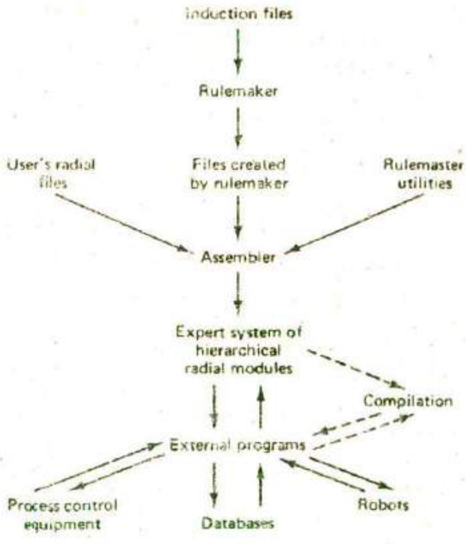


Figure 15.13 Rulemaster building system.

which the problem solving process is carried out. These examples are transformed into rules by Rulemaker through an induction process.

KEE (Knowledge Engineering Environment)

KEE is one of the more popular building tools for the development of larger-scale systems. Developed by Intellicorp, this system employs sophisticated representation schemes structured around frames called units. The frames are made up of slots and facets which contain object attribute values, rules, methods, logical assertions, text, or even other frames. The frames are organized into one or more knowledge bases in the form of hierarchical structures which permit multiple inheritance down hierarchical paths. Rules, procedures, and object oriented representation methods are also supported.

Inference is carried out through inheritance, forward-chaining, backward-chaining, or a mixture of these methods. A form of hypothetical reasoning is also provided through different viewpoints which may be explored concurrently. The viewpoints represent different aspects of a situation, views of the same situation taken at different times, hypothetical situations, or alternative courses of action. This feature permits a user to compare competing courses of action or to reason in parallel about partial solutions based on different approaches.

KEE's support environment includes a graphics-oriented debugging package, flexible end-user interfaces using windows, menus, and an explanation capability with graphic displays which can show inference chains. A graphics-based simulation package called SimKit is available at additional cost.

KEE has been used for the development of intelligent user interfaces, genetics, diagnosis and monitoring of complicated systems, planning, design, process control, scheduling, and simulation. The system is LISP based, developed for operation on systems such as Symbolics machines, Xerox 1100s, or TI Explorers. Systems can also be ported to open architecture machines which support Common LISP without extensive modification.

OPS5 System

The OPS5 and other OPS building tools were developed at Carnegie Mellon University in conjunction with DEC during the late 1970s. This system was developed to build the R1/XCON expert system which configures Vax and other DEC minicomputer systems. The system is used to build rule-based production systems which use forward chaining in the inference process (backward and mixed chaining is also possible). The system was written in C language to run on the DEC Vax and other minicomputers. It uses a sophisticated method of indexing rules (the Rete algorithm) to reduce the matching times during the match-select-execute cycle. Examples of OPS5 rules were given above in Section 15.2, and a description of the Rete match algorithm was given in Section 10.6.

15.7 SUMMARY

Expert and other knowledge-based systems are usually composed of at least a knowledge base, an inference engine, and some form of user interface. The knowledge base, which is separate from the inference and control components, contains the expert knowledge coded in some form such as production rules, networks of frames or other representation scheme. The inference engine manipulates the knowledge structures in the knowledge base to perform a type of symbolic reasoning and draw useful conclusions relating to the current task. The user interface provides the means for dialog between the user and system. The user inputs commands, queries, and responses to system messages, and the system, in turn, produces various messages for the user. In addition to these three components, most systems have an editor for use in creating and modifying the knowledge base structures and an explanation module which provides the user with explanations of how a conclusion was reached or why a piece of knowledge is needed. A few systems also have some learning capability and a case history file with which [illegible] record complete consultation traces.

A variety of expert system architectures have been constructed including rule-based systems, frame-based systems, decision tree (discrimination network) systems, analogical reasoning systems, blackboard architectures, theorem proving systems, and even neural network architectures. These systems may differ in the direction of rule chaining, in the handling of uncertainty, and in the search and pattern matching methods employed. Rule and frame based systems are by far the most popular architectures used.

Since the introduction of the first expert systems in the late 1970s, a number of building tools have been developed. Such tools may be as unsophisticated as a bare high level language or as comprehensive as a complete shell development environment. A few representative building tools have been described and some general characteristics of tools for developers were given.

The acquisition of expert knowledge for knowledge-based systems remains one of the main bottlenecks in building them. This has led to a new discipline called knowledge engineering. Knowledge engineers build systems by eliciting knowledge from experts, coding that knowledge in an appropriate form, validating the knowledge, and ultimately constructing a system using a variety of building tools.

EXERCISES

15.1. What are the main advantages in keeping the knowledge base separate from the control module in knowledge-based systems?

15.2. Why is it important that an expert system be able to explain the why and how questions related to a problem solving session?

15.3. Give an example of the use of metaknowledge in expert systems inference.

15.4. Describe and compare the different types of problems solved by four of the earliest expert systems DENDRAL, MYCIN, PROSPECTOR, and R1.

15.5. Identify and describe two good application areas for expert systems within a university environment.

15.6. How do rules in PROLOG differ from general production system rules?

15.7. Make up a small knowledge-base of facts and rules using the same syntax as that used in Figure 15.2 except that they should relate to an office working environment.

15.8. Name four different types of selection criteria that might be used to select the most relevant rules for firing in a production system.

15.9. Describe a method in which rules could be grouped or organized in a knowledge base to reduce the amount of search required during the matching part of the inference cycle.

15.10. Using the knowledge base of Problem 15.7, simulate three match-select-execute cycles for a query which uses several rules and/or facts.

15.11. Explain the difference between forward and backward chaining and under what conditions each would be best to use for a given set of problems.

15.12. Under what conditions would it make sense to use both forward and backward chaining? Give an example where both are used.

15.13. Explain why you think associative networks were never very popular forms of knowledge representations in expert systems architectures.

15.14. Suppose you are diagnosing automobile engines using a system having a frame type of architecture similar to PIP. Show how a trigger condition might be satisfied for the distributor ignition system when it is learned that the spark at all spark plugs is weak.

15.15. Give the advantages of expert system architectures based on decision trees over those of production rules. What are the main disadvantages?

15.16. Two of the main problems in validating the knowledge contained in the knowledge bases of expert systems are related to completeness and consistency, that is, whether or not a system has an adequate breadth of knowledge to solve the class of problems it was intended to solve and whether or not the knowledge is consistent. Is it easier to check decision tree architectures or production rule systems for completeness and consistency? Give supporting information for your conclusions.

15.17. Give three examples of applications for which blackboard architectures are well suited.

15.18. Give three examples of applications for which the use of analogical architectures would be suitable in expert systems.

15.19. Consider a simple fully connected neural network containing three input nodes and a single output node. The inputs to the network are the eight possible binary patterns 000, 001, . . . , 111. Find weights w_i for which the network can differentiate between the inputs by producing three distinct outputs.

15.20. For the preceding problem, draw projection vectors on the unit circle for the eight different inputs using the weights determined there.

15.21. Explain how uncertainty is propagated through a chain of rules during a consultation with an expert system which is based on the MYCIN architecture.

15.22. Select a problem domain that requires some special expertise and consult with an expert in the domain to learn how he or she solves typical problems. After collecting enough knowledge to solve a small subset of problems, create rules which could be used in a knowledge base to solve the problems. Test the use of the rules on a few problems which have been suggested by the expert and then get his or her confirmation.

15.23. Relate each of the heuristics given by Penny Nii in Section 15.5 to a real expert system solving problem.

15.24. Discuss how each of the features of expert system building tools given in Section 15.6 can affect the performance of the systems developed.

15.25. Obtain a copy of an expert system building tool such as Personal Consultant Plus and create an expert system to diagnose automobile engine problems. Consult with a mechanic to see if your completed system is reasonably good.