15 UNDERSTANDING NATURAL LANGUAGE

But it must be recognized that the notion "probability of a sentence" is an entirely useless one under any known interpretation of this term...

—NOAM CHOMSKY, (1965)

I understand a fury in your words,
But not the words.

—WILLIAM SHAKESPEARE, *Othello*

They have been at a great feast of languages,
and stolen the scraps.

—WILLIAM SHAKESPEARE, *Love's Labour's Lost*

I wish someone would tell me what "Ditty wah ditty" means.

—ARTHUR BLAKE

15.0 The Natural Language Understanding Problem

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

Consider, as an example, the following lines from Shakespeare's *Sonnet XVIII*:

> Shall I compare thee to a summer's day?
> Thou art more lovely and more temperate:
> Rough winds do shake the darling buds of May,
> And summer's lease hath all too short a date:

We cannot understand these lines through a simplistic, literal treatment of meaning. Instead, we must address such issues as:

1. What were Shakespeare's intentions in writing? We must know about human love and its social conventions to begin to understand these lines. Or was Shakespeare just trying to get something to his publisher so he could be paid?
2. Why did Shakespeare compare his beloved to a summer's day? Does he mean that she is 24 hours long and can cause sunburn or that she makes him feel the warmth and beauty of summer?
3. What inferences does the passage require? Shakespeare's intended meaning does not reside explicitly in the text; it must be inferred using metaphors, analogies, and background knowledge. For instance, how do we come to interpret the references to rough winds and the brevity of summer as lamenting the shortness of human life and love?
4. How does metaphor shape our understanding? The words are not mere references to explicit objects such as blocks on a table: the poem's meaning is in the selective attribution of properties of a summer's day to the beloved. Which properties are attributed, and which are not, and above all, why are some properties important while others are ignored?
5. Must a computer-based text-to-speech system know something about the iambic pentameter? How could a computer summarize what this poem is "about," or retrieve it intelligently from a corpus of poetry?

We cannot merely chain together the dictionary meanings of Shakespeare's words and call the result understanding. Instead, we must employ a complex process of capturing the word patterns, parsing the sentences, constructing a representation of the semantic meanings, and interpreting these meanings in light of our knowledge of the problem domain.

Our second example is part of a web ad for a faculty position in computer science.

> The Department of Computer Science of the University of New Mexico. . . is conducting a search to fill two tenure-track positions. We are interested in hiring people with interests in:
>
> Software, including analysis, design, and development tools. . .
>
> Systems, including architecture, compilers, networks. . .
>
> ...
>
> Candidates must have completed a doctorate in. . .
>
> The department has internationally recognized research programs in adaptive computation, artificial intelligence, . . . and enjoys strong research collaborations with the Santa Fe Institute and several national laboratories. . .

Several questions arise in understanding this job ad:

1. How does the reader know that this ad is for a faculty position, when only "tenure-track" is explicitly stated? For how long are people hired as tenure-track?
2. What software and software tools are required for working in a university environment, when none were explicitly mentioned? Cobol, Prolog, UML? A person would need a lot of knowledge about university teaching and research to understand these expectations.
3. What do internationally recognized programs and collaborations with interesting institutions have to do with a university job ad?
4. How could a computer summarize what this ad is about? What must it know to intelligently retrieve this ad from the web for a job-hunting PhD candidate?

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

This chapter provides an introduction to the problems of natural language understanding and the computational techniques developed for their solution. Although in this chapter we focus primarily on the understanding of text, speech-understanding and generation systems must also solve these problems, as well as the additional difficulties associated with the recognition and disambiguation of words grounded in a particular context.

Early AI programs, because of the knowledge required to understand unconstrained language, made progress by restricting their focus to *microworlds*, limited applications that required minimal domain knowledge. One of the earliest programs to take this approach was Terry Winograd's SHRDLU (Winograd 1972), which could converse about a *blocks world* consisting of differently shaped and colored blocks and a hand for moving them about, as in Figure 15.1. See also Section 8.4.

SHRDLU could respond to English-language queries such as "What is sitting on the red block?" "What shape is the blue block on the table?" or "Place the green pyramid on the red brick." It could handle pronoun references such as "Is there a red block? Pick it up." It could even understand ellipses, such as "What color is the block on the blue brick? Shape?" Because of the simplicity of the blocks world, it was possible to provide the system with adequate knowledge. Because the blocks world did not involve the more difficult problems of commonsense reasoning such as understanding time, causality, possibilities, or beliefs, the techniques for representing this knowledge were relatively straightforward. In spite of its limited domain, SHRDLU did provide a model for the integration of syntax and semantics and demonstrated that a program with sufficient knowledge of a domain of discourse could communicate meaningfully in natural language.

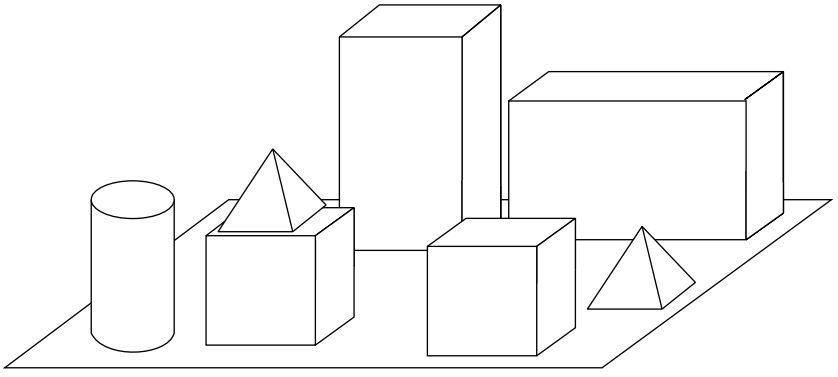


Figure 15.1 A blocks world, adapted from Winograd (1972).

Complementary to the knowledge-intensive component of language understanding just described is to model the patterns and expectations of the language expressions themselves. *Markov Chains* offer a powerful tool for capturing these regularities. In language use, for example, articles and adjectives generally precede nouns rather than follow them and certain nouns and verbs tend to occur together. Markov models can also capture the relationships between language patterns and the worlds they describe.

In Section 15.1 we present a high-level analysis of language understanding. Section 15.2 presents a syntactic analysis; Section 15.3 combines syntax and semantics using augmented transition network parsing. Section 15.4 presents the stochastic approach to capturing regularities in language expressions. Finally, in Section 15.5 we consider several applications where natural language understanding programs are useful: question answering, accessing information in databases, web queries, and text summarization.

15.1 Deconstructing Language: An Analysis

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

1. *Prosody* deals with the rhythm and intonation of language. This level of analysis is difficult to formalize and often neglected; however, its importance is evident in the powerful effect of poetry or religious chants, as well as the role played by rhythm in children's wordplay and the babbling of infants.
2. *Phonology* examines the sounds that are combined to form language. This branch of linguistics is important for computerized speech recognition and generation.

3. *Morphology* is concerned with the components (morphemes) that make up words. These include the rules governing the formation of words, such as the effect of prefixes (un-, non-, anti-, etc.) and suffixes (-ing, -ly, etc.) that modify the meaning of root words. Morphological analysis is important in determining the role of a word in a sentence, including its tense, number, and part of speech.
4. *Syntax* studies the rules for combining words into legal phrases and sentences, and the use of those rules to parse and generate sentences. This is the best formalized and thus the most successfully automated component of linguistic analysis.
5. *Semantics* considers the meaning of words, phrases, and sentences and the ways in which meaning is conveyed in natural language expressions.
6. *Pragmatics* is the study of the ways in which language is used and its effects on the listener. For example, pragmatics would address the reason why "Yes" is *usually* an inappropriate answer to the question "Do you have a watch?"
7. *World knowledge* includes knowledge of the physical world, the world of human social interaction, and the role of goals and intentions in communication. This general background knowledge is essential to understand the full meaning of a text or conversation.

Although these levels of analysis seem natural and are supported by psychological evidence, they are, to some extent, artificial divisions that have been imposed on language. All of these interact extensively, with even low-level intonations and rhythmic variations having an effect on the meaning of an utterance, for example, the use of sarcasm. This interaction is evident in the relationship between syntax and semantics, and although some division along these lines seems essential, the exact boundary is difficult to characterize. For example, sentences such as "They are eating apples" have multiple parsings, resolved only by attention to meaning in context. Syntax also affects semantics, as is seen by the role of phrase structure in interpreting the meaning of a sentence. Although the exact nature of the distinction between syntax and semantics is often debated, both the psychological evidence and its utility in managing the complexity of the problem argue for its retention. We address these deeper issues of language understanding and interpretation again in Chapter 16.

Although the specific organization of natural language understanding programs varies with different philosophies and applications—e.g., a front end for a database, an automatic translation system, a story understanding program—all of them must translate the original sentence into an internal representation of its meaning. Generally, symbol-based natural language understanding follows the stages presented in Figure 15.2.

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

The second stage is *semantic interpretation*, which produces a representation of the meaning of the text. In Figure 15.2 this is shown as a conceptual graph. Other

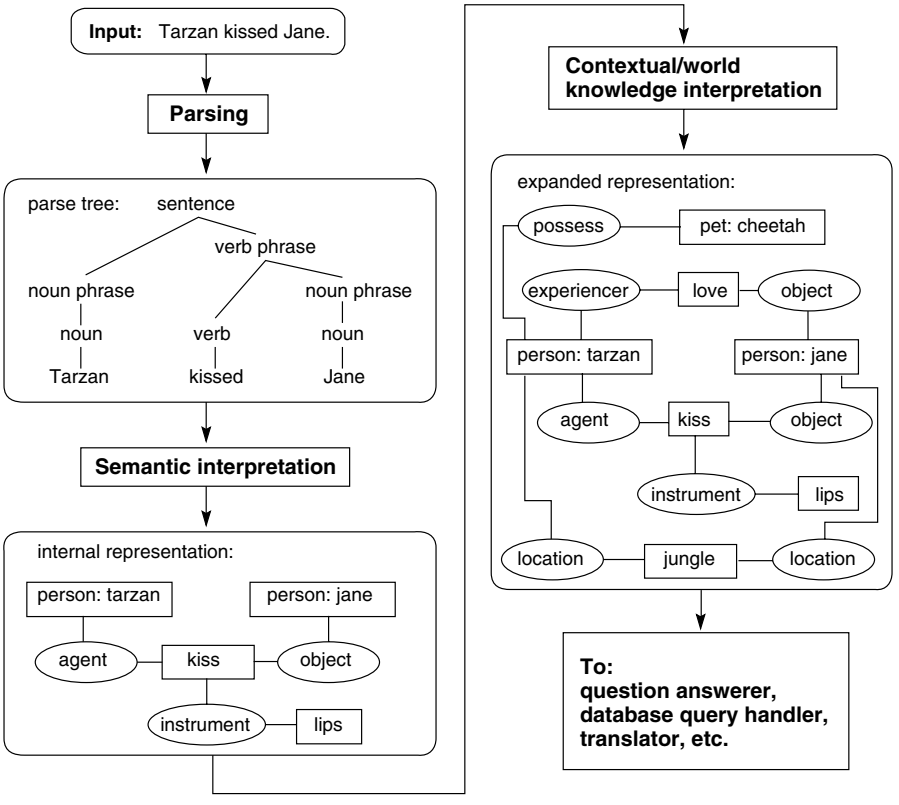


Figure 15.2 Stages in producing an internal representation of a sentence.

representations commonly used include conceptual dependencies, frames, and logic-based representations. Semantic interpretation uses knowledge about the meaning of words and linguistic structure, such as case roles of nouns or the transitivity of verbs. In Figure 15.2, the program used knowledge of the meaning of kiss to add the default value of lips for the instrument of kissing. This stage also performs semantic consistency checks. For example, the definition of the verb kiss may include constraints that the object be a person if the agent is a person, that is, Tarzan kisses Jane and does not (normally) kiss Cheetah.

In the third stage, structures from the knowledge base are added to the internal representation of the sentence to produce an expanded representation of the sentence's meaning. This adds the necessary world knowledge required for complete understanding, such as the facts that Tarzan loves Jane, that Jane and Tarzan live in the jungle, and that Cheetah is Tarzan's pet. This resulting structure represents the meaning of the natural language text and is used by the system for further processing.

In a database front end, for example, the extended structure would combine the representation of the query's meaning with knowledge about the organization of the database.

This could then be translated into an appropriate query in the database language (see Section 15.5.2). In a story understanding program, this extended structure would represent the meaning of the story and be used to answer questions about it (see the discussion of scripts in Chapter 7 and of text summarization in Section 15.5.3).

These stages exist in most (non probabilistic) natural language understanding systems, although they may or may not correspond to distinct software modules. For example, many programs do not produce an explicit parse tree but generate an internal semantic representation directly. Nevertheless, the tree is implicit in the parse of the sentence. *Incremental parsing* (Allen 1987) is a commonly used technique in which a fragment of the internal representation is produced as soon as a significant part of the sentence is parsed. These fragments are combined into a complete structure as the parse proceeds. These fragments are also used to resolve ambiguities and guide the parser.

15.2 Syntax

15.2.1 Specification and Parsing Using Context-Free Grammars

Chapter 3 introduced the use of *rewrite rules* to specify a grammar. The rules listed below define a grammar for simple transitive sentences such as "The man likes the dog." The rules are numbered for reference.

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

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

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

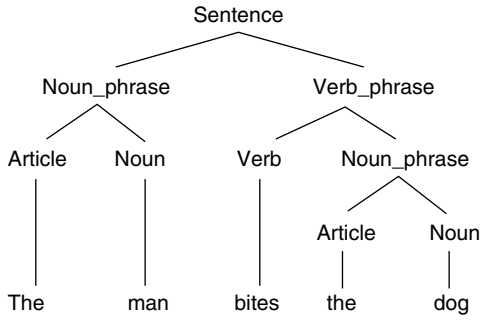


Figure 15.3 Parse tree for the sentence The man bites the dog.

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

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

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

A derivation can be represented as a tree structure, known as a *parse tree*, in which each node is a symbol from the set of rules of the grammar. The tree's interior nodes are nonterminals; each node and its children correspond, respectively, to the left- and right-hand side of a rule in the grammar. The leaf nodes are terminals and the sentence symbol is the root of the tree. The parse tree for "The man bites the dog" appears in Figure 15.3.

Not only does the existence of a derivation or parse tree prove that a sentence is legal in the grammar, but it also determines the structure of the sentence. The *phrase structure* of the grammar defines the deeper linguistic organization of the language. For example, the breakdown of a sentence into a noun_phrase and a verb_phrase specifies the

relation between an action and its agent. This phrase structure plays an essential role in semantic interpretation by defining intermediate stages in a derivation at which semantic processing may take place, as seen in Section 14.3.

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

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

The problem of selecting the correct rule at any stage of the parse is handled either by allowing the parser to set backtrack pointers and return to the problem situation if an incorrect choice was made (see Section 3.2) or by using look-ahead to check the input stream for features to help determine the proper rule to apply. With either approach, we must take care to control the complexity of execution while guaranteeing a correct parse.

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

Because parsing is particularly important in the processing of programming languages as well as natural language, researchers have developed a number of different parsing algorithms, including both top-down and bottom-up strategies. Although a complete survey of parsing algorithms is beyond the scope of this chapter, we do consider several approaches to parsing in some detail. In the next section we show the Earley parser, an important polynomial-time parser based on dynamic programming. In Section 15.3, we introduce semantic constraints to parsers, with *transition network* parsers. These parsers are not sufficiently powerful for the semantic analysis of natural language, but they do form the basis for *augmented transition networks*, which have proved to be a useful and powerful tool in natural language work.

15.2.2 The Earley Parser: Dynamic Programming Revisited

Dynamic programming (DP) was originally proposed by Richard Bellman (1956) and presented with several examples in Section 4.1. The idea is straightforward: when addressing a complex problem that can be broken down into multiple interrelated subproblems, save

partial solutions as they are generated so that they can be reused in the continuing solution process. This approach is sometimes called memoizing the subproblem results for reuse.

There are many examples of dynamic programming in pattern matching, for example, in determining a difference measure between two strings of bits or characters. The overall difference between the strings will be a function of the differences between their specific components. An example of this is a spell checker suggesting words that are "close" to your misspelled word (Section 4.1). Another example of DP is to recognize words in speech understanding as a function of the possible phonemes from an input stream. As phonemes are recognized (with associated probabilities), the most appropriate word is often a function of the combined conjoined probabilistic measures of the individual phones (see Section 13.X). In this section, we use DP to show the Earley parser determining whether strings of words make up syntactically correct sentences. Our pseudo-code is adapted from that of Jurafsky and Martin (2008).

The parsing algorithms of Section 15.2.1 are often implemented with a recursive, depth-first, and left-to-right search of possible acceptable syntactic structures. This search approach can mean that many of the possible acceptable partial parses of the first (left-most) components of the string are repeatedly regenerated. This revisiting of early partial solutions within the full parse structure is the result of later backtracking requirements of the search and can become exponentially expensive and in larger parses. Dynamic programming provides an efficient alternative where partial parses, once generated, are saved for reuse in the overall final parse of a string of words. The first DP-based parser was created by Earley (1970).

Memoization and Dotted Pairs

In parsing with Earley's algorithm the memoization of partial solutions (partial parses) is done with a data structure called a *chart*. This is why the Earley approach to parsing is often called *chart parsing*. The chart is generated through the use of dotted grammar rules.

The dotted grammar rule provides a representation that indicates, in the chart, the state of the parsing process at any given time. Every dotted rule falls into one of three categories, depending on whether the dot's position is at the beginning, somewhere in the middle, or at the end of the right hand side, RHS, of the grammar rule. We refer to these three categories as the *initial*, *partial*, or *completed* parsing stages, respectively:

Initial prediction: Symbol → • RHS_unseen
Partial parse: Symbol → RHS_seen • RHS_unseen
Completed parse: Symbol → RHS_seen •

In addition, there is a natural correspondence between states containing different dotted rules and the edges of the parse tree(s) produced by the parse. Consider the following very simple grammar, where terminal symbols are surrounded by quotes, as in "mary":

Sentence → Noun Verb
Noun → "mary"
Verb → "runs"

As we perform a top-down, left-to-right parse of this sentence, the following sequence of states is produced:

Sentence → • Noun Verb	*predict*: Noun followed by a Verb
Noun → • mary	*predict*: mary
Noun → mary •	*scanned*: mary
Sentence → Noun • Verb	*completed*: Noun; *predict*: Verb
Verb → • runs	*predict*: runs
Verb → runs •	*scanned*: runs
Sentence → Noun Verb •	*completed*: Verb, *completed*: sentence

Note that the scanning and completing procedures deterministically produce a result. The prediction procedure describes the possible parsing rules that can apply to the current situation. Scanning and prediction create the states in the parse tree of Figure 15.4.

Earley's algorithm operates by generating top-down and left-to-right predictions of how to parse a given input. Each prediction is recorded as a state containing all the relevant information about the prediction, where the key component of each state is a dotted rule. (A second implementation component will be introduced in the next section.) All of the predictions generated after examining a particular word of the input are collectively referred to as the *state set*.

For a given input sentence with n words, w_1 to w_n, a total of n + 1 state sets are generated: [S_0, S_1, …, S_n]. The initial state set S_0 contains those predictions that are made before examining any input words, S_1 contains predictions made after examining w_1, and so on. We refer to the entire collection of state sets as the *chart* produced by the parser. Figure 15.4 illustrates the relationship between state set generation and the examination of input words. Although, traditionally, the sets of states that make up each component of the parse are called state sets, the order of the generation of these states is important. Thus we will call each component of the chart the *state list*, and describe it as [$State_1$, $State_2$, …, $State_n$]. This also works well with the Prolog implementation in the auxiliary materials, where the state lists will be maintained as Prolog lists. Finally, we describe each state of the state list as a sequence of specific symbols enclosed by brackets, for example, ($ → • S).

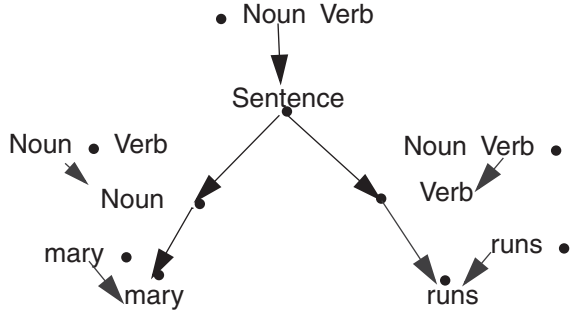


Figure 15.4 The relationship of dotted rules to the generation of a parse tree.

Now consider Earley's algorithm parsing the simple sentence mary runs, using the grammar above. The algorithm begins by creating a dummy start state, ($ → • S), that is the first member of state list S_0. This state represents the prediction that the input string can be parsed as a sentence, and it is inserted into S_0 prior to examining any input words. A successful parse produces a final state list S_n, which contains the state ($ → S •).

Beginning with S_0, the parser executes a loop in which each state, S_i, in the current state list is examined in order and used to generate new states. Each new state is generated by one of three procedures that are called the predictor, scanner, and completer. The appropriate procedure is determined by the dotted rule in state S, specifically by the grammar symbol (if any) following the dot in the rule.

In our example, the first state to be examined contains the rule ($ → • S). Since the dot is followed by the symbol S, this state is "expecting" to see an instance of S occur next in the input. As S is a nonterminal symbol of the grammar, the predictor procedure generates all states corresponding to a possible parse of S. In this case, as there is only one alternative for S, namely that S → Noun Verb, only one state, (S → • Noun Verb), is added to S_0. As this state is expecting a part of speech, denoted by the nonterminal symbol Noun following the dot, the algorithm examines the next input word to verify that prediction. This is done by the scanner procedure, and since the next word matches the prediction, mary is indeed a Noun, the scanner generates a new state recording the match: (Noun → mary •). Since this state depends on input word w_1, it becomes the first state in state list S_1 rather than being added to S_0. At this point the chart, containing two state lists, looks as follows, where after each state we name the procedure that generated it:

S_0:	[($ → • S),	dummy start state
	(S → • Noun Verb)]	predictor
S_1:	[(Noun → mary •)]	scanner

Each state in the list of states S_0 has now been processed, so the algorithm moves to S_1 and considers the state (Noun → mary •). Since this is a completed state, the completer procedure is applied. For each state expecting a Noun, that is, has the • Noun pattern, the completer generates a new state that records the discovery of a Noun by advancing the dot over the Noun symbol. In this case, the completer produces the state (S → • Noun Verb) in S_0 and generates the new state (S → Noun • Verb) in the list S_1. This state is expecting a part of speech, which causes the scanner to examine the next input word w_2. As w_2 is a Verb, the scanner generates the state (Verb → runs •) and adds it to S_2, resulting in the following chart:

S_0:	[($ → • S),	start
	(S → • Noun Verb)]	predictor
S_1:	[(Noun → mary •),	scanner
	(S → Noun • Verb)]	completer
S_2:	[(Verb → runs •)]	scanner

Processing the new state S_2, the completer advances the dot in (S → Noun • Verb) to produce (S → Noun Verb •), from which the completer generates the state ($ → S •) signifying a successful parse of a sentence. The final chart for mary runs, with three state lists, is:

S_0:	[($ → • S),	start
	(S → • Noun Verb)]	predictor
S_1:	[(Noun → mary •),	scanner
	(S → Noun • Verb)]	completer
S_2:	[(Verb → runs •)]	scanner
	(S → Noun Verb •),	completer
	($ → S •)]	completer

15.2.2.2 The Earley Algorithm: Pseudo-code

To represent computationally the state lists produced by the dotted pair rules above, we create indices to show how much of the right hand side of a grammar rule has been parsed. We first describe this representation and then offer pseudo-code for implementing it within the Earley algorithm.

Each state in the state list is augmented with an index indicating how far the input stream has been processed. Thus, we extend each state description to a (*dotted rule [i, j]*) representation where the [i, j] pair denotes how much of right hand side, RHS, of the grammar rule has been seen or parsed to the present time. For the right hand side of a parsed rule that includes zero or more seen and unseen components indicated by the •, we have (A → Seen • Unseen, [i,j]), where i is the start of Seen and j is the position of • in the word sequence.

We now add indices to the parsing states discussed earlier for the sentence mary runs:

($ → • S, [0, 0])	produced by predictor, i = j = 0, nothing parsed
(Noun → mary •, [0,1])	scanner sees w_1 between word indices 0 and 1
(S→ Noun • Verb, [0,1])	completer has seen Noun (mary) between 0 and 1
(S → Noun Verb •, [0,2])	completer has seen sentence S between 0 and 2

Thus, the state indexing pattern shows the results produced by each of the three state generators using the dotted rules along with the word index w_i.

To summarize, the three procedures for generating the states of the state list are: predictor generating states with index [j, j] going into chart[j], scanner considering word w_{j+1} to generate states indexed by [j, j+1] into chart[j+1], and completer operating on rules with index [i, j], i < j, adding a state entry to chart[j]. Note that a state from the dotted-rule, [i, j] always goes into the state list chart[j]. Thus, the state lists include chart[0], ..., chart[n] for a sentence of n words. Now that we have presented the indexing scheme for representing the chart, we give the pseudo-code for the Earley parser.

```
function EARLEY-PARSE(words, grammar)  returns chart
begin
     chart := empty
     ADDTOCHART(($ → • S, [0, 0]), chart[0])              % dummy start state
     for i from 0 to LENGTH(words) do
          for each state in chart[i] do
               if rule_rhs(state) = … • A … and A is not a part of speech
                then PREDICTOR(state)
                else if rule_rhs(state) = … • L …          % L is part of speech
                     then SCANNER(state)
                     else COMLETER(state)                   % rule_rhs = RHS •
end

procedure PREDICTOR((A → … • B …, [i,  j]))
begin
     for each (B → RHS) in grammar do
          ADDTOCHART((B →  • RHS, [j,  j]), chart[j])
end

procedure SCANNER((A → … • L …, [i,  j]))
begin
     if (L → word[j]) is_in grammar
          then ADDTOCHART((L → word[j] •, [j,  j + 1]), chart[j + 1])
end

procedure COMPLETER((B → …  •, [j, k]))
begin
     for each (A → … • B …, [i, j]) in chart[j] do
          ADDTOCHART((A → … B • …, [i, k]), chart[k])
end

procedure ADDTOCHART(state, state-list)
begin
     if state is not in state-list
     then ADDTOEND(state, state-list)
end
```

Our first example, the Earley parse of the sentence Mary runs, was intended to be simple but illustrative, with the very detailed presentation of the state lists and their indices. A more complex - and ambiguous - sentence, John called Mary from Denver is presented in the auxiliary materials with code for implementing a parse using the Earley algorithm in both the Prolog and Java languages. The ambiguity of this sentence is reflected in the production of two different parses that redflecting that ambiguity. The selection of one of these parses is acomplished by running the backward component of the dynamic programming based Earley parser.

15.3 Transition Network Parsers and Semantics

To this point we have offered representations and algorithms that support symbol based syntactic anslysis. The analysis of syntactic relationships, even when constrained for context sensitive parsing (e.g., noun-verb agreement), does not consider semantic relationships. In this section we introduce *transition networks* to address this issue.

15.3.1 Transition Network Parsers

A transition network parser represents a grammar as a set of finite-state machines or *transition networks*. Each network corresponds to a single nonterminal in the grammar. Arcs are labeled with either terminal or nonterminal symbols. Each path through the network, the start state to the final state, corresponds to some rule for that nonterminal; the sequence of arc labels on the path is the sequence of symbols on the right-hand side of the rule. The grammar of Section 15.2.1 is represented by the networks of Figure 15.5. When there is more than one rule for a nonterminal, the corresponding network has multiple paths from the start to the goal, e.g., the rules noun_phrase ↔ noun and noun_phrase ↔ article noun are captured by alternative paths through the noun_phrase network.

Finding a successful transition through the network for a nonterminal corresponds to the replacement of that nonterminal by the right-hand side of a grammar rule. For example, to parse a sentence, a transition network parser must find a transition through the sentence network. It begins in the start state ($S_{initial}$) and takes the noun_phrase transition and then the verb_phrase transition to reach the final state (S_{final}). This is equivalent to replacing the original sentence symbol with noun_phrase verb_phrase.

In order to cross an arc, the parser examines its label. If the label is a terminal symbol, the parser checks the input stream to see whether the next word matches the arc label. If it does not match, the transition cannot be taken. If the arc is labeled with a nonterminal symbol, the parser retrieves the network for that nonterminal and recursively attempts to find a path through it. If the parser fails to find a path through this network, the top-level arc cannot be traversed. This causes the parser to backtrack and attempt another path through the network. Thus, the parser tries to find a path through the sentence network; if it succeeds, the input string is a legal sentence in the grammar.

Consider the simple sentence Dog bites, as analyzed and illustrated in Figure 15.6:

1. The parser begins with the sentence network and tries to move along the arc labeled noun_phrase. To do so, it retrieves the network for noun_phrase.
2. In the noun_phrase network, the parser first tries the transition marked article. This causes it to branch to the network for article.
3. It fails to find a path to the finish node of the article network because the first word, "Dog," matches neither of the arc labels. The parser fails and backtracks to the noun_phrase network.
4. The parser attempts to follow the arc labeled noun in the noun_phrase network and branches to the network for noun.

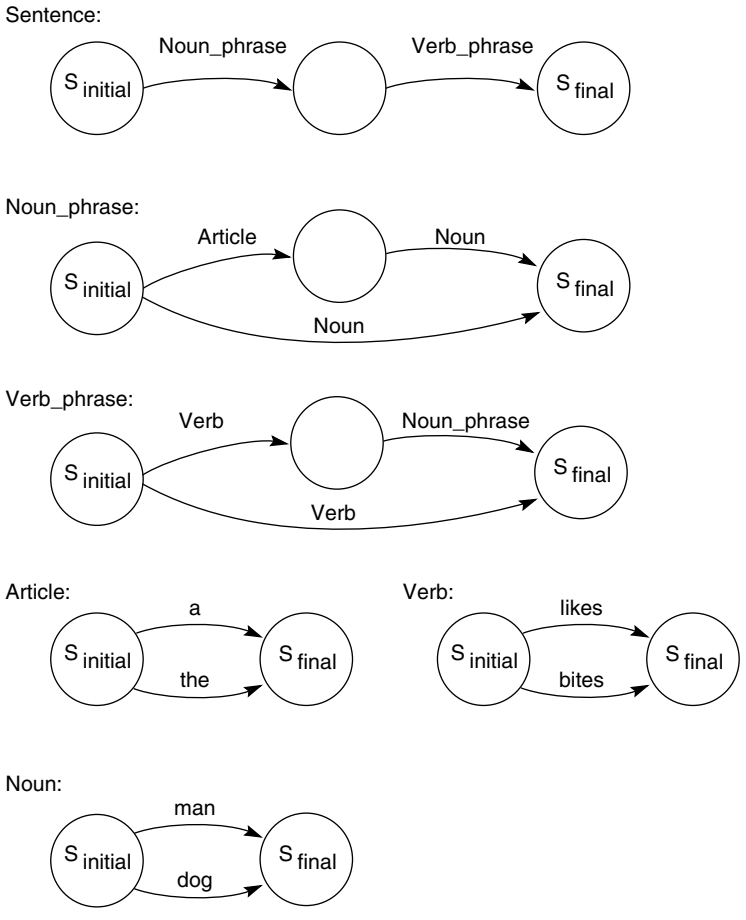


Figure 15.5 Transition network definition of a simple English grammar.

5. The parser successfully crosses the arc labeled "dog," because this corresponds to the first word of the input stream.
6. The noun network returns success. This allows the arc labeled noun in the noun_phrase network to be crossed to the final state.
7. The noun_phrase network returns success to the top-level network, allowing the transition of the arc labeled noun_phrase.
8. A sequence of similar steps is followed to parse the verb_phrase portion of the sentence.

Pseudo-code for a transition network parser appears below. It is defined using two mutually recursive functions, parse and transition. Parse takes a grammar symbol as argument: if the symbol is a terminal, parse checks it against the next word in the input

stream. If it is a nonterminal, parse retrieves the transition network associated with the symbol and calls transition to find a path through the network. To parse a sentence, call parse(sentence).

```
function parse(grammar_symbol);
begin
   save pointer to current location in input stream;
   case
      grammar_symbol is a terminal:
         if grammar_symbol matches the next word in the input stream
            then return (success)
            else begin
               reset input stream;
               return (failure)
            end;
      grammar_symbol is a nonterminal:
         begin
            retrieve the transition network labeled by grammar symbol;
            state := start state of network;
            if transition(state) returns success
            then return (success)
            else begin
               reset input stream;
               return (failure)
            end
         end
   end
end.

function transition (current_state);
begin
   case
      current_state is a final state:
         return (success)
      current_state is not a final state:
         while there are unexamined transitions out of current_state
            do begin
               grammar_symbol := the label on the next unexamined transition;
               if parse(grammar_symbol) returns (success)
                  then begin
                     next_state := state at end of the transition;
                     if transition(next_state) returns success;
                           then return (success)
                  end
            end
      return (failure)
   end
end.
```

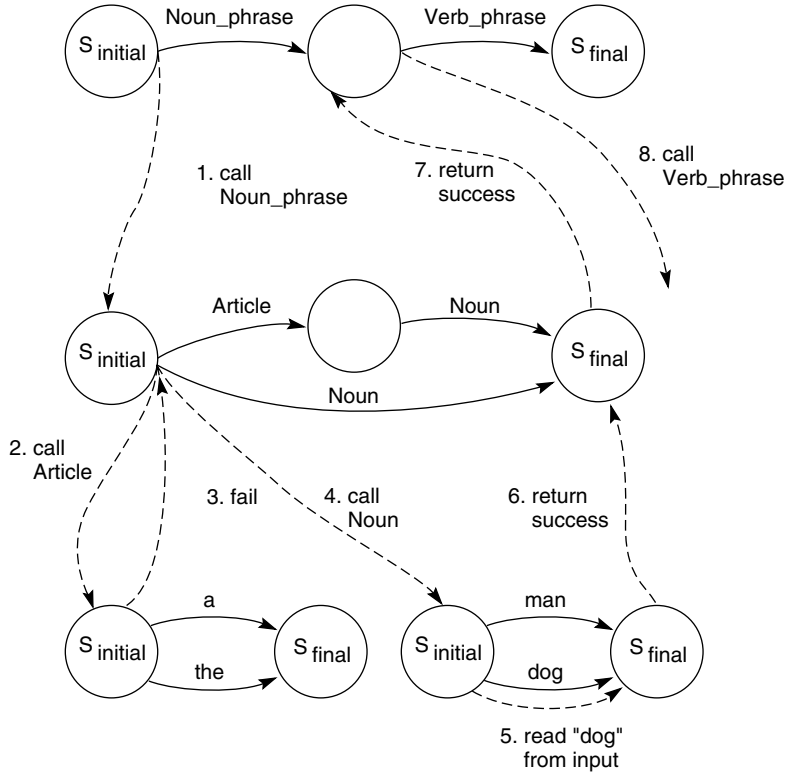


Figure 15.6 Trace of a transition network parse of the sentence Dog bites.

Transition takes a state in a transition network as its argument and tries to find a path through that network in a depth-first fashion. Because the parser may make a mistake and require a backtrack, parse retains a pointer to the current location in the input stream. This allows the input stream to be reset to this location in the event the parser backtracks.

This transition network parser determines whether a sentence is grammatically correct, but it does not construct a parse tree. This may be accomplished by having the functions return a subtree of the parse tree instead of the symbol success. Modifications that would accomplish this are:

1. Each time the function parse is called with a terminal symbol as argument and that terminal matches the next symbol of input, it returns a tree consisting of a single leaf node labeled with that symbol.
2. When parse is called with a nonterminal, grammar_symbol, it calls transition. If transition succeeds, it returns an ordered set of subtrees (described below). Parse combines these into a tree whose root is grammar_symbol and whose children are the subtrees returned by transition.

3. In searching for a path through a network, transition calls parse on the label of each arc. On success, parse returns a tree representing a parse of that symbol. Transition saves these subtrees in an ordered set and, on finding a path through the network, returns the ordered set of parse trees corresponding to the sequence of arc labels on the path.

15.3.2 The Chomsky Hierarchy and Context-Sensitive Grammars

In Section 15.3.1, we defined a small subset of English using a *context-free grammar*. A context-free grammar allows rules to have only a single nonterminal on their left-hand side. Consequently, the rule may be applied to any occurrence of that symbol, regardless of its context. Though context-free grammars have proved to be a powerful tool for defining programming languages and other formalisms in computer science, there is reason to believe that they are not powerful enough, by themselves, to represent the rules of natural language syntax. For example, consider what happens if we add both singular and plural nouns and verbs to the grammar of Section 15.2.1:

noun ↔ men
noun ↔ dogs
verb ↔ bites
verb ↔ like

The resulting grammar will parse sentences like "The dogs like the men", but it also accepts "A men bites a dogs." The parser accepts these sentences because the rules don't use context to determine when singular and plural forms need be coordinated. The rule defining a sentence as a noun_phrase followed by a verb_phrase does not require that the noun and verb agree on number, or that articles agree with nouns.

Conext free languages can be extended to handle these situations, but a more natural approach is for the grammer to be context sensitive, where the components of the parse tree are designed to constrain each other. Chomsky (1965) first proposed a world of hierarchical and ever more powerful grammers (Hopcroft and Ullman 1979). At the bottom of this hierarchy is the class of *regular languages*, whose grammar may be defined using a finite-state machine, Section 3.1. Regular languages have many uses in computer science, but they are not powerful enough to represent the syntax of most programming languages.

The *context-free languages* are above the regular languages in the Chomsky hierarchy. Context-free languages are defined using rewrite rules such as in Section 15.2.1; context-free rules may only have one nonterminal symbol on their left-hand side. Transition network parsers are able to parse the class of context-free languages. It is interesting to note that if we *do not* allow recursion in a transition network parser i.e., arcs may be labeled only with terminal symbols and transitions may not "call" another network, then the class of languages that may be so defined corresponds to regular expressions. Thus, regular languages are a proper subset of the context-free languages.

The *context-sensitive* languages form a proper superset of the context-free languages. These are defined using *context-sensitive grammars* which allow more than one symbol

on the left-hand side of a rule and make it possible to define a context in which that rule can be applied. This ensures satisfaction of global constraints such as number agreement and other semantic checks. The only restriction on context-sensitive grammar rules is that the right-hand side be at least as long as the left-hand side (Hopcroft and Ullman 1979).

A fourth class, forming a superset of the context-sensitive languages, is the class of *recursively enumerable* languages. Recursively enumerable languages may be defined using unconstrained production rules; because these rules are less constrained than context-sensitive rules, the recursively enumerable languages are a proper superset of the context-sensitive languages. This class is not of interest in defining the syntax of natural language, although it is important in the theory of computer science. The remainder of this section focuses on English as a context-sensitive language.

A simple context-free grammar for sentences of the form article noun verb that enforces number agreement between article and noun and subject and verb is given by:

sentence ↔ noun_phrase verb_phrase
noun_phrase ↔ article number noun
noun_phrase ↔ number noun
number ↔ singular
number ↔ plural
article singular ↔ a singular
article singular ↔ the singular
article plural ↔ some plural
article plural ↔ the plural
singular noun ↔ dog singular
singular noun ↔ man singular
plural noun ↔ men plural
plural noun ↔ dogs plural
singular verb_phrase ↔ singular verb
plural verb_phrase ↔ plural verb
singular verb ↔ bites
singular verb ↔ likes
plural verb ↔ bite
plural verb ↔ like

In this grammar, the nonterminals singular and plural offer constraints to determine when different article, noun, and verb_phrase rules can be applied, ensuring number agreement. A derivation of the sentence "The dogs bite" using this grammar is given by:

sentence.
noun_phrase verb_phrase.
article plural noun verb_phrase.
The plural noun verb_phrase.
The dogs plural verb_phrase.
The dogs plural verb.
The dogs bite.

Similarly, we can use context-sensitive grammars to perform checks for semantic agreement. For example, we could disallow sentences such as "Man bites dog" by adding a nonterminal, act_of_biting, to the grammar. This nonterminal could be checked in the rules to prevent any sentence involving "bites" from having "man" as its subject.

Though context-sensitive grammars can define language structures that cannot be captured using context-free grammars, they have a number of disadvantages for the design of practical parsers:

1. Context-sensitive grammars increase drastically the number of rules and nonterminals in the grammar. Imagine the complexity of a context-sensitive grammar that would include number, person, and all the other forms of agreement required by English.
2. They obscure the phrase structure of the language that is so clearly represented in the context-free rules.
3. By attempting to handle more complicated checks for agreement and semantic consistency in the grammar itself, they lose many of the benefits of separating the syntactic and semantic components of language.
4. Context-sensitive grammars do not address the problem of building a semantic representation of the meaning of the text. A parser that simply accepts or rejects sentences is not sufficient; it must return a useful representation of the sentence's semantic meaning.

In the auxiliary materials for this book we present both context-free and context sensitive parsers and sentence generators. The control regime for these Prolog programs is depth-first recursive descent. Next, we examine *augmented transition networks* (ATNs), an extension of transition networks that can define context-sensitive languages but has several advantages over context-sensitive grammars in the design of parsers.

15.3.3 Semantics: Augmented Transition Network Parsers

An alternative to context-sensitive grammars is to retain the simpler structure of context-free grammar rules but augment these rules with attached procedures that perform the necessary contextual tests. These procedures are executed when a rule is invoked in parsing. Rather than using the grammar to describe such notions as number, tense, and person, we represent these as *features* attached to terminals and nonterminals of the grammar. The procedures attached to the rules of the grammar access these features to assign values and perform the necessary tests. Grammars that use augmentations of context-free grammars to implement context sensitivity include *augmented phrase structure grammars* (Heidorn 1975, Sowa 1984), *augmentations of logic grammars* (Allen 1987), and the *augmented transition network* (ATN).

In this section we present ATN parsing and outline the design of a simple ATN parser for sentences about the "dogs world" introduced in Section 15.2.1. We address the first two steps of Figure 15.2: creation of a parse tree and its use to construct a representation

of the sentence's meaning. We use conceptual graphs in this example, although ATN parsers can also be used with semantic net, script, frame, or logic-based representations.

Augmented transition networks extend transition networks by allowing procedures to be attached to the arcs of the networks. An ATN parser executes these attached procedures when it traverses the arcs. The procedures may assign values to grammatical features and perform tests, causing a transition to fail if certain conditions (such as number agreement) are not met. These procedures also construct a parse tree, which is used to generate an internal semantic representation of the sentence's meaning.

We represent both terminals and nonterminals as identifiers (e.g., verb, noun_phrase) with attached features. For example, a word is described using its morphological root, along with features for its part of speech, number, person, etc. Nonterminals in the grammar are similarly described. A noun phrase is described by its article, noun, number, and person. Both terminals and nonterminals can be represented using framelike structures with named slots and values. The values of these slots specify grammatical features or pointers to other structures. For example, the first slot of a sentence frame contains a pointer to a noun phrase definition. Figure 15.7 shows the frames for the sentence, noun_phrase, and verb_phrase nonterminals in our simple grammar.

Individual words are represented using similar structures. Each word in the dictionary is defined by a frame that specifies its part of speech (article, noun, etc.), its morphological root, and its significant grammatical features. In our example, we are only checking for number agreement and only record this feature. More sophisticated grammars indicate person and other features. These dictionary entries may also indicate the conceptual graph definition of the word's meaning for use in semantic interpretation. The complete dictionary for our grammar appears in Figure 15.8.

Figure 15.9 presents an ATN for our grammar, with pseudo-code descriptions of the tests performed at each arc. Arcs are labeled with both nonterminals of the grammar (as in Figure 15.5) and numbers; these numbers are used to indicate the function attached to each arc. These functions must run successfully in order to traverse the arc.

When the parser calls a network for a nonterminal, it creates a new frame for that non-terminal. For example, on entering the noun_phrase network it creates a new noun_phrase frame. The slots of the frame are filled by the functions for that network. These slots may be assigned values of grammatical features or pointers to components of

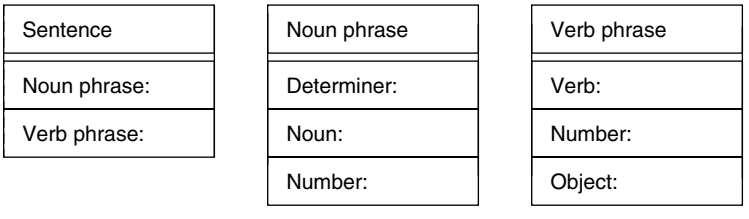


Figure 15.7 Structures representing the sentence, noun phrase, and verb phrase nonterminals of the grammar.

the syntactic structure (e.g., a verb_phrase can consist of a verb and a noun_phrase). When the final state is reached, the network returns this structure.

When the network traverses arcs labeled noun, article, and verb, it reads the next word from the input stream and retrieves that word's definition from the dictionary. If the word is not the expected part of speech, the rule fails; otherwise the definition frame is returned.

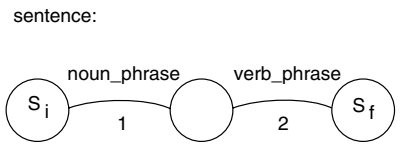
In Figure 15.9 frames and slots are indicated using a Frame.Slot notation; e.g., the number slot of the verb frame is indicated by VERB.NUMBER. As the parse proceeds, each function builds and returns a frame describing the associated syntactic structure. This structure includes pointers to structures returned by lower-level networks. The top-level

Word	Definition	Word	Definition
a	PART_OF_SPEECH: article ROOT: a NUMBER: singular	like	PART_OF_SPEECH: verb ROOT: like NUMBER: plural
bite	PART_OF_SPEECH: verb ROOT: bite NUMBER: plural	likes	PART_OF_SPEECH: verb ROOT: like NUMBER: singular
bites	PART_OF_SPEECH: verb ROOT: bite NUMBER: singular	man	PART_OF_SPEECH: noun ROOT: man NUMBER: singular
dog	PART_OF_SPEECH: noun ROOT: dog NUMBER: singular	men	PART_OF_SPEECH: noun ROOT: man NUMBER: plural
dogs	PART_OF_SPEECH: noun ROOT: dog NUMBER: plural	the	PART_OF_SPEECH: article ROOT: the NUMBER: plural or singular

Figure 15.8 Dictionary entries for a simple ATN.

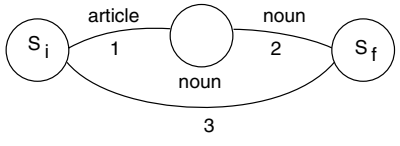
sentence function returns a sentence structure representing the parse tree for the input. This structure is passed to the semantic interpreter. Figure 15.10 shows the parse tree that is returned for the sentence "The dog likes a man."

The next phase of natural language processing takes the parse tree, such as that of Figure 15.10, and constructs a semantic representation of the domain knowledge and meaning content of the sentence.




```
function sentence-1;
   begin
      NOUN_PHRASE := structure returned by
              noun_phrase network;

      SENTENCE.SUBJECT := NOUN_PHRASE;
   end.
```

```
function sentence-2;
   begin
      VERB_PHRASE := structure returned by
               verb_phrase network;

      if NOUN_PHRASE.NUMBER =
                                    VERB_PHRASE.NUMBER
         then begin
            SENTENCE.VERB_PHRASE := VERB_PHRASE;
            return SENTENCE
         end
         else fail
   end.
```



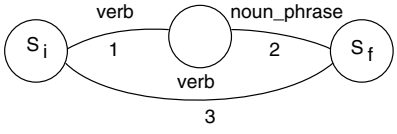

```
function noun_phrase-1;
   begin
      ARTICLE := definition frame for next word of input;

      if ARTICLE.PART_OF_SPEECH=article
         then NOUN_PHRASE.DETERMINER := ARTICLE
         else fail
   end.
```

```
function noun_phrase-2;
   begin
      NOUN := definition frame for next word of input;

      if NOUN.PART_OF_SPEECH=noun and
              NOUN.NUMBER agrees with
                    NOUN_PHRASE.DETERMINER.NUMBER
         then begin
            NOUN_PHRASE.NOUN := NOUN;
            NOUN_PHRASE.NUMBER := NOUN.NUMBER
            return NOUN_PHRASE
         end
         else fail
   end.
```

Figure 15.9 An ATN grammar that checks number agreement and builds a parse tree.

```
function noun_phrase-3
   begin
      NOUN := definition frame for next word of input;

      if NOUN.PART_OF_SPEECH=noun
         then begin
            NOUN_PHRASE.DETERMINER := unspecified;
            NOUN_PHRASE.NOUN := NOUN
            NOUN_PHRASE.NUMBER := NOUN.NUMBER
         end
         else fail
   end.
```

verb_phrase:




```
function verb_phrase-1
   begin
      VERB := definition frame for next word of input;

      if VERB.PART_OF_SPEECH=verb
         then begin
            VERB_PHRASE.VERB := VERB;
            VERB_PHRASE.NUMBER := VERB.NUMBER;
         end;
   end.


function verb_phrase-2
   begin
      NOUN_PHRASE := structure returned by
                                    noun_phrase network;

      VERB_PHRASE.OBJECT := NOUN_PHRASE;
      return VERB_PHRASE
   end.


function verb_phrase-3
   begin
      VERB := definition frame for next word of input;

      if VERB.PART_OF_SPEECH=verb
         then begin
            VERB_PHRASE.VERB := VERB;
            VERB_PHRASE.NUMBER := VERB.NUMBER;
            VERB_PHRASE.OBJECT := unspecified;
            return VERB_PHRASE;
         end;
   end.
```

Figure 15.9 (cont d) An ATN grammar that checks number agreement and builds a parse tree.

15.3.4 Combining Syntax and Semantic Knowledge with ATNs

The semantic interpreter constructs a representation of the input string's meaning by beginning at the root, or sentence node, and traversing the parse tree. At each node, it recursively interprets the children of that node and combines the results into a single conceptual graph; this graph is passed up the tree. For example, the semantic interpreter builds a representation of the verb_phrase by recursively building representations of the

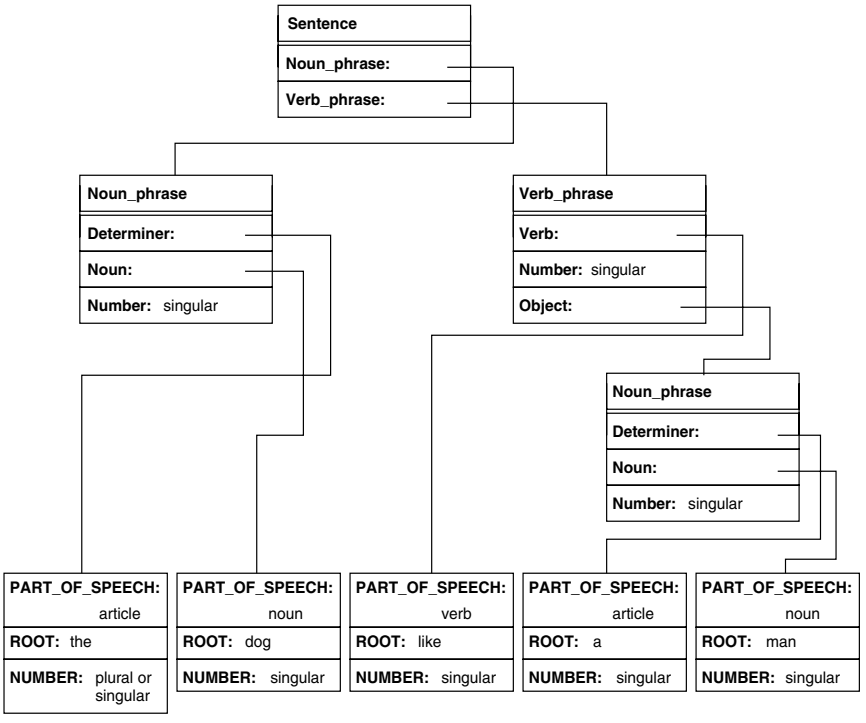


Figure 15.10 Parse tree for the sentence The dog likes a man returned by an ATN parser.

node's children, verb and noun_phrase, and joining these to form an interpretation of the verb phrase. This is passed to the sentence node and combined with the representation of the subject.

Recursion stops at the terminals of the parse tree. Some of these, such as nouns, verbs, and adjectives, cause concepts to be retrieved from the knowledge base. Others, such as articles, do not directly correspond to concepts in the knowledge base but qualify other concepts in the graph.

The semantic interpreter in our example uses a knowledge base for the "dogs world." Concepts in the knowledge base include the objects dog and man and the actions like and bite. These concepts are described by the type hierarchy of Figure 15.11.

In addition to concepts, we must define the relations that will be used in our conceptual graphs. For this example, we use the following concepts:

agent links an act with a concept of type animate. agent defines the relation between an action and the animate object causing the action.

experiencer links a state with a concept of type animate. It defines the relation between a mental state and its experiencer.

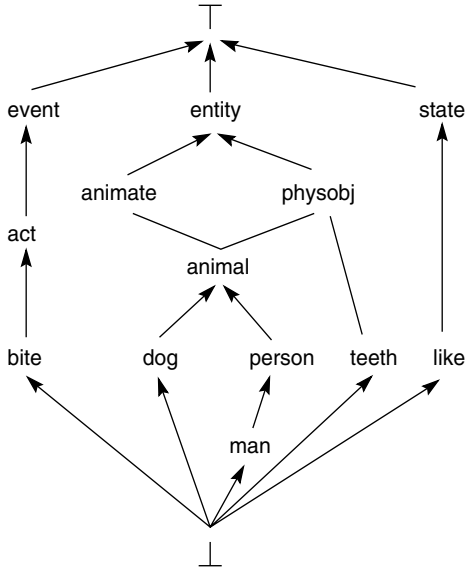


Figure 15.11 Type hierarchy used in dogs world example.

instrument links an act with an entity and defines the instrument used in an action.

object links an event or state with an entity and represents the verb–object relation.

part links concepts of type physobj and defines the relation between whole and part.

The verb plays a particularly important role in building an interpretation, as it defines the relationships between the subject, object, and other components of the sentence. We represent each verb using a *case frame* that specifies:

1. The linguistic relationships (agent, object, instrument, and so on) appropriate to that particular verb. Transitive verbs, for example, have an object; intransitive verbs do not.
2. Constraints on the values that may be assigned to any component of the case frame. For example, in the case frame for the verb "bites," we have asserted that the agent must be of the type dog. This causes "Man bites dog" to be rejected as semantically incorrect.
3. Default values on components of the case frame. In the "bites" frame, we have a default value of teeth for the concept linked to the instrument relation.

The case frames for the verbs like and bite appear in Figure 15.12.

We define the actions that build a semantic representation with rules or procedures for each potential node in the parse tree. Rules for our example are described as pseudo-code

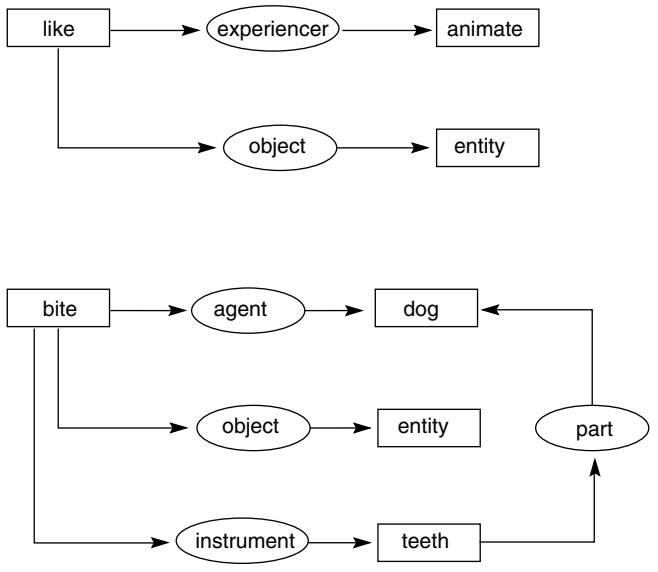


Figure 15.12 Case frames for the verbs like and bite.

procedures. In each procedure, if a specified join or other test fails, that interpretation is rejected as semantically incorrect:

```
procedure sentence;

  begin
    call noun_phrase to get a representation of the subject;
    call verb_phrase to get a representation of the verb_phrase;
    using join and restrict, bind the noun concept returned for the subject to
      the agent of the graph for the verb_phrase
  end.

procedure noun_phrase;

  begin
    call noun to get a representation of the noun;
    case
      the article is indefinite and number singular: the noun concept is generic;
      the article is definite and number singular: bind marker to noun concept;
      number is plural: indicate that the noun concept is plural
    end case
  end.
```

```
procedure verb_phrase;

  begin
    call verb to get a representation of the verb;
    if the verb has an object
      then begin
        call noun_phrase to get a representation of the object;
        using join and restrict, bind concept for object to object of the verb
      end
    end.

procedure verb;

  begin
    retrieve the case frame for the verb
  end.

procedure noun;

  begin
    retrieve the concept for the noun
  end.
```

Articles do not correspond to concepts in the knowledge base but determine whether their noun concept is generic or specific. We have not discussed the representation of plural concepts; refer to Sowa (1984) for their treatment as conceptual graphs.

Using these procedures, along with the concept hierarchy of Figure 15.11 and the case frames of Figure 15.12, we trace the actions of the semantic interpreter in building a semantic representation of the sentence "The dog likes a man" from the parse tree of Figure 15.10. This trace appears in Figure 15.13.

The actions taken in the trace are (numbers refer to Figure 15.13):

1. Beginning at the sentence node, call sentence.
2. sentence calls noun_phrase.
3. noun_phrase calls noun.
4. noun returns a concept for the noun dog (1 in Figure 15.13).
5. Because the article is definite, noun_phrase binds an individual marker to the concept (2) and returns this concept to sentence.
6. sentence calls verb_phrase.
7. verb_phrase calls verb, which retrieves the case frame for like (3).
8. verb_phrase calls noun_phrase, which then calls noun to retrieve the concept for man (4).
9. Because the article is indefinite, noun_phrase leaves this concept generic (5).

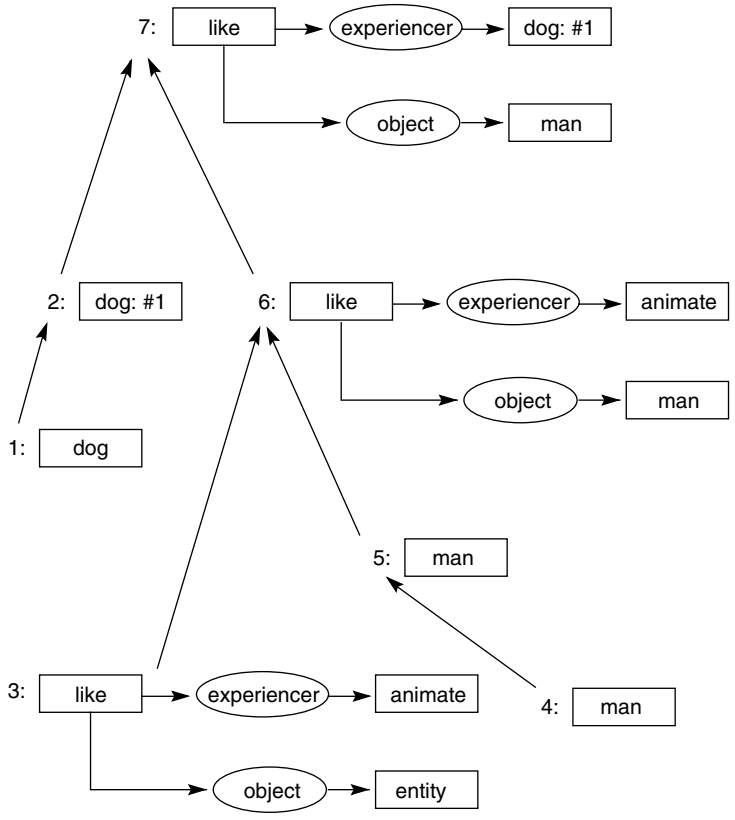


Figure 15.13 Construction of a semantic representation from the parse tree of Figure 15.10.

10. The verb_phrase procedure restricts the entity concept in the case frame and joins it with the concept for man (6). This structure is returned to sentence.
11. sentence joins concept dog: #1 to the experiencer node of the case frame (7).

This conceptual graph represents the meaning of the sentence.

Language generation is a related problem addressed by natural language understanding programs. The generation of English sentences requires the construction of a semantically correct output from an internal representation of meaning. For example, the agent relation indicates a subject–verb relationship between two concepts. Simple approaches allow the appropriate words to be plugged into stored sentence *templates*. These templates are patterns for sentences and fragments, such as noun phrases and prepositional phrases. The output is constructed by walking the conceptual graph and combining these fragments. More sophisticated approaches to language generation use transformational grammars to map meaning into a range of possible sentences (Winograd 1972, Allen 1987).

In the auxiliary materials for this book we have build a full recursive-descent semantic net parser in Prolog. This parser uses graph operators, such as join and restrict, to constrain the semantic net representations attached to the leaf nodes of the parse tree.

In Section 15.5 we will show how a program can build an internal representations of language phenomena. These representations are used by programs in a number of ways, depending on applications, sevewral of which we present. But first, in Section 15.4, we present stochastic approaches for capturing the patterns and regularities of language.

15.4 Stochastic Tools for Language Understanding

In Section 15.1 we discussed a natural decomposition of sentence structures that supported language understanding. In Sections 15.2 and 15.3 we noted that the semantic and knowledge intensive aspects of language could be supported by word and sentence-level representational structures. In this section we introduce stochastic tools that support, at these same levels, pattern based analysis of structures enabling language comprehension.

15.4.1 Introduction: Statistical Techniques in Language Analysis

Statistical language techniques are methods which arise when we view natural language as a random process. In everyday parlance, randomness suggests lack of structure, definition, or understanding. However, viewing natural language as a random process *generalizes* the deterministic viewpoint. That is, statistical (or stochastic) techniques can accurately model both those parts of language which are well defined as well as those parts which indeed do have some degree of randomness.

Viewing language as a random process allows us to redefine many of the basic problems within natural language understanding in a rigorous, mathematical manner. It is an interesting exercise, for example, to take several sentences, say the previous paragraph including periods and parentheses, and to print out these same words and language symbols ordered by a random number generator. The result will make very little sense. It is interesting to note (Jurafsky and Martin 2008) that these same pattern-based constraints operate on many levels of linguistic analysis, including acoustic patterns, phonemic combinations, the analysis of grammatical structure, and so on. As an example of the use of stochastic tools, we first consider the problem of part-of-speech tagging.

Most people are familiar with this problem from grammar class. We want to label each word in a sentence as a noun, verb, preposition, adjective, and so on. In addition, if the word is a verb, we may want to know if it is active, passive, transitive, or intransitive. If the word is a noun, whether it is singular or plural and so on. Difficulty arises with words like “swing”. If we say, “front porch swing,” swing is a noun but if we say “swing at the ball” then swing is a verb. We present next a quote from Picasso with its correct part of speech labels:

Art	is	a	lie	that	lets	us	see	the	truth.
Noun	Verb	Article	Noun	Pronoun	Verb	Pronoun	Verb	Article	Noun

To begin our analysis, we first define the problem formally. We have a set of words in our language $S_w = \{w_1,..., w_n\}$, for example {a, aardvark, ... , zygote}, and a set of parts of speech or tags $S_t = \{t_1, ..., t_m\}$. A sentence with n words is a sequence of n random variables $W_1, W_2, ... W_n$. These are called *random* variables because they can take on any of the values in S_w with some probability. The tags, $T_1, T_2, ...,T_n$, are also a sequence of random variables. The value that T_i takes on will be denoted t_i and the value of W_i is w_i. We want to find the sequence of values for these tags which is most likely, given the words in the sentence. Formally, we want to pick $t_1, ..., t_n$ to maximize:

$$p(T_1= t_1, ..., T_n = t_n \mid W_1 = w_1, ..., W_n = w_n)$$

Recall from Section 5.2, that $p(X \mid Y)$ stands for *the probability of X given that Y has occurred*. It is customary to drop reference to the random variables and just write:

$$p(t_1, ..., t_n \mid w_1, ..., w_n) \qquad \text{equation 1}$$

Note that if we knew this probability distribution exactly and if we had enough time to maximize over all possible tag sets, we would always get the best possible set of tags for the words considered. In addition, if there really were only one correct sequence of tags for each sentence, an idea your grammar teacher may have espoused, this probabilistic technique would always find that correct sequence! Thus the probability for the correct sequence would be 1 and that for all other sequences 0. This is what we meant when we said that the statistical viewpoint can generalize the deterministic one.

In reality, because of limited storage space, data, and time, we cannot use this technique and must come up with some type of approximation. The rest of this section deals with better and better ways to approximate equation 1.

First note that we can rewrite equation 1 in a more useful manner:

$$p(t_1, ..., t_n \mid w_1, ..., w_n) = p(t_1, ..., t_n , w_1, ..., w_n) / p(w_1, ..., w_n)$$

and since we maximize this by choosing $t_1, ..., t_n$, we can simplify equation 1 to:

$$p(t_1, ..., t_n, w_1,..., w_n) =$$

$$p(t_1)p(w_1 \mid t_1)p(t_2 \mid t_1,w_1) ... p(t_n \mid w_1, ..., w_n , t_1, ..., t_{n-1}) =$$

$$\prod_{i=1}^{n} p(t_i \mid t_1, ..., t_{i-1}, w_1, ..., w_{i-1})\, p(w_i \mid t_1, ..., t_{i-1}, w_1, ..., w_{i-1}) \qquad \text{equation 2}$$

Notice that equation 2 is equivalent to equation 1.

15.4.2 A Markov Model Approach

In practice, and as seen earlier in our discussion of Chapters 9.3 and 13, it is usually a complex task to maximize equations with probabilities conditioned on many other random

variables, such as we find in equation 2. There are three reasons for this: first, it is difficult to store the probability of a random variable conditioned on many other random variables because the number of possible probabilities increases exponentially with the number of conditioning variables. Secondly, even if we could store all of the probability values, it is often difficult to estimate their values. Estimation is usually done empirically by counting the number of occurrences of an event in a hand-tagged training corpus and thus, if an event occurs only a few times in that training set, we will not get a good estimate of its probability. That is, it is easier to estimate p(cat | the) than p(cat | The dog chased the) since there will be fewer occurrences of the latter in the training set. Finally, finding the chain of tags that maximizes structures like equation 2 would take too long, as will be shown next.

First, we make some useful approximations of equation 2. The first rough attempt is:

$p(t_i \mid t_1, \ldots, t_{i-1}, w_1, \ldots, w_{i-1})$ approaches $p(t_i \mid t_{i-1})$

and

$p(w_i \mid t_1, \ldots, t_{i-1}, w_1, \ldots, w_{i-1})$ approaches $p(w_i \mid t_i)$.

These are *first-order Markov assumption*, as seen in Section 9.3, where it is assumed that the present thing being considered is dependent only on the immediately preceeding thing, i.e., it is independent of things in the more distant past.

Plugging these approximations back into equation 2, we get

$$\prod_{i=1}^{n} p(t_i \mid t_{i-1})\, p(w_i \mid t_i) \qquad \text{equation 3}$$

Equation 3 is straightforward to work with because its probabilities can be easily estimated and stored. Recall that equation 3 is just an estimate of $P(t_1,\ldots, t_n \mid w_1,\ldots, w_n)$ and we still need to maximize it by choosing the tags, i.e., $t_1,\ldots, t_n$. Fortunately, there is a dynamic programming (DP) algorithm called the Viterbi algorithm (Viterbi 1967, Forney 1973; for DP see Sections 4.1 and 15.2, for Viterbi, Section 10.3) which will allow us to do this. The Viterbi algorithm calculates the probability of t^2 tag sequences for each word in the sentence where t is the number of possible tags. For a particular step, the tag sequences under consideration are of the following form:

```
article article     {best tail}
article verb        {best tail}
...
article noun        {best tail}
...
...
noun article        {best tail}
...
noun  noun          {best tail}
```

where {best tail} is the most likely sequence of tags found dynamically for the last $n - 2$ words for the given $n - 1$ tag.

There is an entry in the table for every possible value for tag number $n - 1$ and tag number n (hence we have the t^2 tag sequences). At each step, the algorithm finds the maximal probabilities and adds one tag to each best tail sequence. This algorithm is guaranteed to find the tag sequence which maximizes equation 3 and it runs in $O(t^2 s)$, where t is the number of tags and s is the number of words in the sentence. If $p(t_i)$ is conditioned on the last n tags rather than the last two, the Viterbi Algorithm will take $O(t^n s)$. Thus we see why conditioning on too many past variables increases the time taken to find a maximizing value.

Fortunately, the approximations used in equation 3 work well. With about 200 possible tags and a large training set to estimate probabilities, a tagger using these methods is about 97% accurate, which approaches human accuracy. The surprising accuracy of the Markov approximation along with its simplicity makes it useful in many applications. For example, most speech recognition systems use what is called the *trigram* model to provide some "grammatical knowledge" to the system for predicting words the user has spoken. The trigram model is a simple Markov model which estimates the probability of the current word conditioned on the previous two words. It uses the Viterbi algorithm and other techniques just described. For more detail on this and related techniques see Jurasky and Martin (2008).

15.4.3 A Decision Tree Approach

An obvious problem with the Markov approach is that it considers only local context. If instead of tagging words with simple parts of speech, we wish to do things like identify an agent, identify an object, or decide whether verbs are active or passive, then a richer context is required. The following sentence illustrates this problem:

The policy announced in December by the President guarantees lower taxes.

In fact, the *President* is the agent but a program using a Markov model would likely identify the *policy* as agent and *announced* as an active verb. We can imagine that a program would get better at probabilistically choosing the agent of this type of sentence if it could ask questions like, "Is the current noun inanimate?" or "Does the word *by* appear a few words before the noun under consideration?"

Recall that the tagging problem is equivalent to maximizing equation 2, i.e.,

$$\prod_{i=1}^{n} p(t_i \mid t_1, \ldots, t_{i-1}, w_1, \ldots, w_{i-1})\, p(w_i \mid t_1, \ldots, t_{i-1}, w_1, \ldots, w_{i-1}).$$

Theoretically, considering a larger context involves simply finding better estimates for these probabilities. This suggests that we might want to use answers to the grammatical questions above to refine the probabilities.

There are several ways we can address this issue. First, we can combine the Markov approach with the parsing techniques presented in the first three sections of this chapter. A

second method allows us to find probabilities conditioned on *yes* or *no* questions with the ID3 algorithm (presented in detail in Section 10.3 and built in Lisp in the auxiliary material) or some equivalent algorithm. ID3 trees have the added bonus that out of a very large set of possible questions, they will choose only those which are good at refining probability estimates. For more complicated natural language processing tasks such as parsing, ID3 based trees are often preferred over Markov models. We next describe how to use ID3 to construct a decision tree for use in parsing.

Recall that in the above section we asked the question "Is the current noun inanimate?" We can ask questions like this only if we know which words are animate and which words are inanimate. In fact there is an automated technique which can assign words to these types of classes for us. The technique is called *mutual information clustering*. The mutual information shared between two random variables X and Y is defined as follows:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log_2 \frac{p(x, y)}{p(x)p(y)}$$

To do mutual information clustering over a vocabulary of words, we start by putting each word in the vocabulary into a distinct set. At each step, we compute the average mutual information between sets using a bigram, that is a next word model, and a merge of two word sets is chosen which minimizes the loss in average mutual information for all classes.

For example, if initially we have the words cat, kitten, run, and green, at the first step of the algorithm, we have the sets:

{cat} {kitten} {run} {green}.

It is likely that the probability of the next word, given that the previous word was cat, is about equal to the probability of the next word, given that the previous word was kitten. In other words:

p(eats | cat) is about the same as p(eats | kitten)
p(meows | cat) is about the same as p(meows | kitten)

Thus, if we let X1, X2, Y1, and Y2 be random variables such that:

X1 = {{cat}, {kitten}, {run}, {green}}
Y1 = word following X1
X2 = {{cat, kitten}, {run}, {green}}
Y2 = word following X2,

then the mutual information between X2 and Y2 is not much less than the mutual information between X1 and Y1, thus cat and kitten will likely be combined. If we continue this procedure until we have combined all possible classes, we get a binary tree.

Then, *bit codes* can be assigned to words based on the branches taken within the tree that reaches the leaf node that has that word in it. This reflects the semantic meaning of the word. For example:

```
cat    = 01100011
kitten = 01100010
```

Furthermore, we might find that "noun-like" words will be all those words that have a 1 in the leftmost bit and that words which most likely represent inanimate objects may be those whose 3rd bit is a 1.

This new encoding of dictionary words allows the parser to ask questions more effectively. Note that the clustering does not take context into account so that even though "book" may be clustered as a "noun-like" word, we will want our model to tag it as a verb when it is found in the phrase "book a flight."

15.4.4 Probabilistic Approaches to Parsing

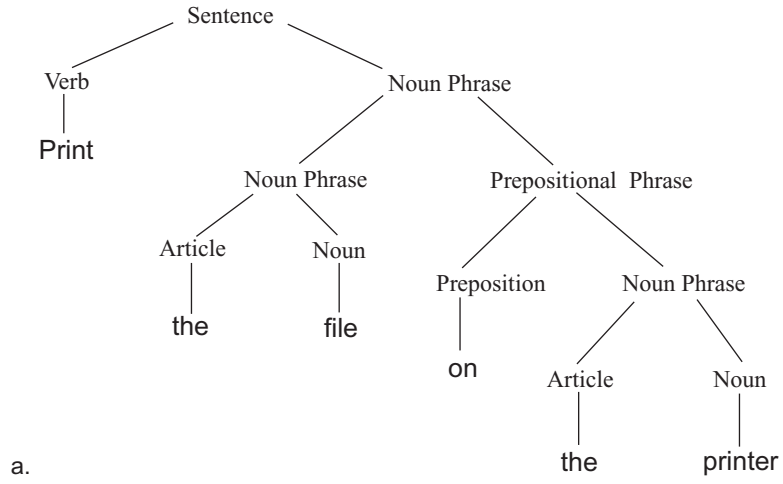
Stochastic techniques have already been used in many domains of computational linguistics and there is still a great deal of opportunity to apply them to areas which have resisted traditional, symbolic approaches.

The use of statistical methods in parsing was first motivated by the problem of ambiguity. Ambiguity arises from the fact that there are often several possible parses for a given sentence and we need to choose which parse might be the best one. For example, the sentence Print the file on the printer can be parsed using either of the two trees presented in Figure 15.14.

In situations such as this, grammar rules alone are not sufficient for choosing the correct parse. In the Print the file on the printer case we need to consider some information about context and semantics. In fact, the primary use of stochastic techniques in the parsing domain is to help resolve ambiguities. In the current example, we can use the same tool used in part of speech tagging, the ID3 algorithm. ID3 assists us in predicting the probability that a parse is correct based on semantic questions about the sentence. In the case when there is some syntactic ambiguity in the sentence, we can then choose that parse which has the highest probability of being correct. As usual, this technique requires a large *training corpus* of sentences with their correct parses.

Recently, people in the statistical natural language modeling community have become more ambitious and have tried to use statistical techniques without a grammar to do parsing. Although the details of grammarless parsing are beyond the scope of this book, suffice it to say that it is related more to pattern recognition than to the traditional parsing techniques covered earlier in this chapter.

Grammarless parsing has been quite successful. In experiments comparing a traditional grammar-based parser with a grammarless one on the task of parsing the same set of sentences, the grammar-based parser achieved a score, using a popular metric, the *crossing-brackets* measure, of 69% and the grammarless parser 78% (Magerman 1994). These results are good, although not outstanding. More importantly, the grammar in the tradi-



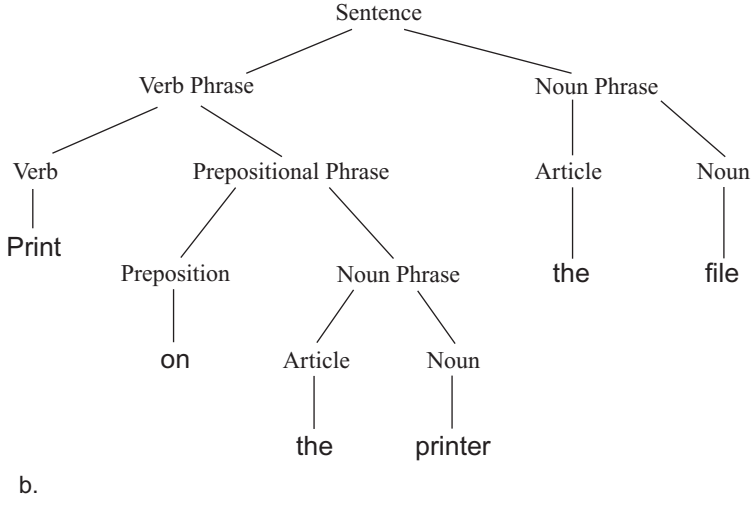


Figure 14.14 Two different parses of Print the file on the printer.

tional parser was developed meticulously by a trained linguist over the course of about ten years, while the grammarless parser used essentially no hard-coded linguistic information, only sophisticated mathematical models which could infer the needed information from the training data. For more on grammarless parsing and related issues see Manning and Schutze (1999).

Speech understanding, converting speech to text, and handwriting recognition are three further areas which have a long history of using stochastic methods for modeling

language. The most commonly used statistical method in these areas is the trigram model for next word prediction. The strength of this model is in its simplicity: it predicts the next word based on the last two words. Recently, there has been work in the statistical language community to maintain the simplicity and ease of use of this model while incorporating grammatical constraints and longer distance dependencies. This new approach uses what are called *grammatical trigrams*. The grammatical trigrams are informed by basic associations between pairs of words (i.e., subject–verb, article–noun, and verb–object). Collectively, these associations are called a *link grammar*. The link grammar is much simpler and easier to construct than the traditional grammars used by linguists and works well with probabilistic methods.

Berger et al. (1994) describe a statistical program, Candide, which translates French text to English text. Candide uses both statistics and information theory to develop a probability model of the translation process. It trains only on a large corpus of French and English sentence pairs and gets results comparable to and in some cases better than Systran (Berger et al. 1994), a commercial translation program. Of particular interest is the fact that the Candide system does no traditional parsing in the translation process. Instead it uses the grammatical trigrams and link grammars just mentioned.

15.4.5 Probabilistic Context-Free Parsers

In this section we present two different probabilistic context-free parsers, structure-based and lexicalized. The structure-based parser uses probability measures for each grammatical parse-rule that occurs. The probability of each parse rule is a function of that rule happening combined with the probabilities of its constituents.

We demonstrate the *structure based probabilistic parser* by extending the set of context-free grammar rules of Section 15.2 with probability measures:

1. sentence ↔ noun_phrase verb_phrase p(s) = p(r1) p(np) p(vp)
2. noun_phrase ↔ noun p(np) = p(r2) p(noun)
3. noun_phrase ↔ article noun p(np) = p(r3) p(article) p(noun)
4. verb_phrase ↔ verb p(vp) = p(r4) p(verb)
5. verb_phrase ↔ verb noun_phrase p(vp) = p(r5) p(verb) p(np)
6. article ↔ a p(article) = p(a)
7. article ↔ the p(article) = p(the)
8. noun ↔ man p(noun) = p(man)
9. noun ↔ dog p(noun) = p(dog)
10. verb ↔ likes p(noun) = p(likes)
11. verb ↔ bites p(noun) = p(bites)

The probability measures for the individual words can be taken from their probability of occurring within a particular corpus of English sentences. The probability measure for each grammar rule, the $p(r_i)$, can be determined by how often that grammar rule occurs within the sentences of a language corpus. Of course, the example here is simple - it is intended to characterize this type of probabilistic context-free parser.

The second example, the *probabilistic lexicalized context free parser*, will also be demonstrated by extending the grammar rules of Section 15.2. In this situation we want to take account of multiple aspects of the sentence. First we can have bigram or trigram probabilities of language use. Again we will be dependent on an appropriate language corpus to obtain bigram or trigram measures. As with our previous example we will also use probability measures, obtained from a language corpus, for the occurrence of each of the parse rules. Finally, we will have probabilistic measures of particular nouns and verb going together. For example, if the subject noun is dog. the probability that the verb will be bites. Note that this measure, even though taken from a language corpus, can also enforce noun-verb agreement.

We call our parser *lexicalized*, because we are able to focus on the probabilities of specific patterns of words occurring together. For simplicity we present our parser for bigrams only:

1. sentence ↔ noun_phrase(noun) verb_phrase(verb)
   p(sentence) = p(r1) p(np) p(vp) p(verb | noun)
2. noun_phrase ↔ noun
   p(np) = p(r2) p(noun)
3. noun_phrase ↔ article noun
   p(np) = p(r3) p(article) p(noun) p(noun | article)
4. verb_phrase ↔ verb
   p(vp) = p(r4) p(verb)
5. verb_phrase ↔ verb noun_phrase
   p(vp) = p(r5) p(verb) p(noun_phrase(noun)) p(noun |verb)
6. article ↔ a p(article) = p(a)
7. article ↔ the p(article) = p(the)
8. noun ↔ man p(noun) = p(man)

. p(man | a)
p(man | the)
etc.
p(likes | man)
etc.

Each of the two probabilistic parsers of this section have been built in Prolog in the auxiliary materials for this chapter

There are many other areas where rigorous stochastic language modeling techniques have not yet been tried but where they may yield useful results. Information extraction, or the problem of obtaining a certain amount of concrete information from a written text is one potential application. A second is tom match patterns in voiced speech to link into concepts of a specific knowledge base. Finally, of course, is semantic web searching. Further details on stochastic approaches to natural language processing can be found in Jurafsky and Martin (2008) and Manning and Schutze (1999).

We end Chapter 15 with several examples of computational language use.

15.5 Natural Language Applications

15.5.1 Story Understanding and Question Answering

An interesting test for natural language understanding technology is to write a program that can read a story or other piece of natural language text and answer questions about it. In Chapter 7 we discussed some of the representational issues involved in story understanding, including the importance of combining background knowledge with the explicit content of the text. As illustrated in Figure 15.2, a program can accomplish this by performing network joins between the semantic interpretation of the input and conceptual graph structures in a knowledge base. More sophisticated representations, such as scripts, Section 7.1.4, can model more complex situations involving events occurring over time.

Once the program has built an expanded representation of the text, it can intelligently answer questions about what it has read. The program parses the question into an internal representation and matches that query against the expanded representation of the story. Consider the example of Figure 15.2. The program has read the sentence "Tarzan kissed Jane" and built an expanded representation.

Assume that we ask the program "Who loves Jane?" In parsing the question, the interrogative, who, what, why, etc., indicates the intention of the question. *Who* questions ask for the agent of the action; *what* questions ask for the object of the action; *how* questions ask for the means by which the action was performed, and so on. The question "Who loves Jane?" produces the graph of Figure 15.15. The agent node of the graph is marked with a ? to indicate that it is the goal of the question. This structure is then joined with the expanded representation of the text. The concept that becomes bound to the person: ? concept in the query graph is the answer to the question: "Tarzan loves Jane."

15.5.2 A Database Front End

The major bottleneck in designing natural language understanding programs is the acquisition of sufficient knowledge about the domain of discourse. Current technology is limited to narrow domains with well-defined semantics. An application area that meets these criteria is the development of natural language front ends for databases. Although databases store enormous amounts of information, that information is highly regular and narrow in scope; furthermore, database semantics are well defined. These features, along with the utility of a database that can accept natural language queries, make database front ends an important application of natural language understanding technology.

The task of a database front end is to translate a question in natural language into a well-formed query in the database language. For example, using the SQL database language as a target (Ullman 1982), the natural language front end would translate the question "Who hired John Smith?" into the query:

```
SELECT MANAGER
FROM MANAGER_OF_HIRE
WHERE EMPLOYEE =  John Smith
```

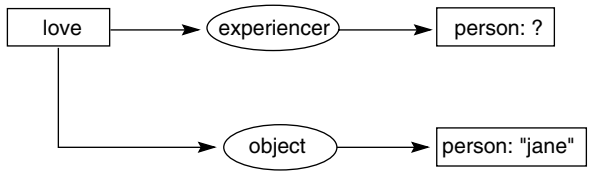


Figure 15.15 Conceptual graph for the question Who loves Jane?

In performing this translation, the program must do more than translate the original query; it must also decide where to look in the database (the MANAGER_OF_HIRE relation), the name of the field to access (MANAGER), and the constraints on the query (EMPLOYEE = 'John Smith'). None of this information was in the original question; it was found in a knowledge base that knew about the organization of the database and the meaning of potential questions.

A *relational database* organizes data in relations across domains of entities. For example, suppose we are constructing a database of employees and would like to access the salary of each employee and the manager who hired her. This database would consist of three *domains*, or sets of entities: the set of managers, the set of employees, and the set of salaries. We could organize these data into two relations, employee_salary, which relates an employee and her salary, and manager_of_hire, which relates an employee and her manager. In a relational database, relations are usually displayed as tables that enumerate the instances of the relation. The columns of the tables are often named; these names are called *attributes* of the relation. Figure 15.16 shows the tables for the employee_salary and the manager_of_hire relations. Manager_of_hire has two attributes, the employee and the manager. The values of the relation are the pairs of employees and managers.

If we assume that employees have a unique name, manager, and salary, then the employee name can be used as a *key* for both the salary and the manager attributes. An attribute is a key for another attribute if it uniquely determines the value of elements for the other attribute. A valid query indicates a target attribute and specifies a value or set of

manager_of_hire:

employee	manager
John Smith	Jane Martinez
Alex Barrero	Ed Angel
Don Morrison	Jane Martinez
Jan Claus	Ed Angel
Anne Cable	Bob Veroff

employee_salary:

employee	salary
John Smith	$35,000.00
Alex Barrero	$42,000.00
Don Morrison	$50,000.00
Jan Claus	$40,000.00
Anne Cable	$45,000.00

Figure 15.16 Two relations in an employee database.

constraints; the database returns the specified values of the target attribute. We can indicate the relationship between keys and other attributes graphically in a number of ways, including *entity-relationship diagrams* (Ullman 1982) and *data flow diagrams* (Sowa 1984). Both of these approaches display the mapping of keys onto attributes using directed graphs.

We can extend conceptual graphs to include diagrams of these relationships (Sowa 1984). The database relation that defines the mapping is indicated by a rhombus, which is labeled with the name of the relation. The attributes of the relation are expressed as concepts in a conceptual graph and the direction of the arrows indicates the mapping of keys onto other attributes. The entity-relation graphs for the employee_salary and manager_of_hire relations may be seen in Figure 15.17.

In translating from English to a formal query, we must determine the record that contains the answer, the field of that record that is to be returned, and the values of the keys that determine that field. Rather than translating directly from English into the database language, we first translate into a more expressive language such as conceptual graphs. This is necessary because many English queries are ambiguous or require additional interpretation to produce a well-formed database query. The use of a more expressive representation language helps this process.

The natural language front end parses and interprets the query into a conceptual graph, as described earlier in this chapter. It then combines this graph with information in the knowledge base using join and restrict operations. In this example, we want to handle queries such as "Who hired John Smith?" or "How much does John Smith earn?" For each potential query, we store a graph that defines its verb, the case roles for that verb, and any relevant entity-relationship diagrams for the question. Figure 15.18 shows the knowledge base entry for the verb "hire."

The semantic interpreter produces a graph of the user's query and joins this graph with the appropriate knowledge base entry. If there is an attached entity relation graph that maps keys into the goal of the question, the program can use this entity relation graph to form a database query. Figure 15.19 shows the query graph for the question "Who hired John Smith?" and the result of joining this with the knowledge base entry from Figure 15.18. It also shows the SQL query that is formed from this graph. Note that the name of the appropriate record, the target field, and the key for the query were not specified in the natural language query. These were inferred by the knowledge base.

In Figure 15.18 the agent and object of the original query were known only to be of type person. To join these with the knowledge base entry for hire, they were first restricted to types manager and employee, respectively. The type hierarchy could thus be used to perform type checking across possible solutions of the original query. If john smith were not of type employee, the question would be invalid and the program could detect this.

Once the expanded query graph is built, the program examines the target concept, flagged with a ?, and determines that the manager_of_hire relation mapped a key onto this concept. Because the key is bound to a value of john smith, the question was valid and the program would form the proper database query. Translation of this entity relationship graph into SQL or some other any other general purpose data base language is straightforward.

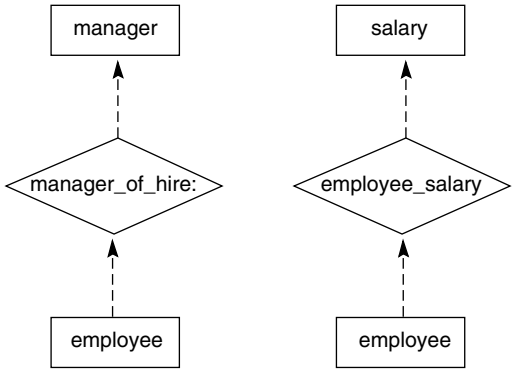


Figure 15.17 Entity-relationship diagrams of the manager_of_hire and employee_salary relations.

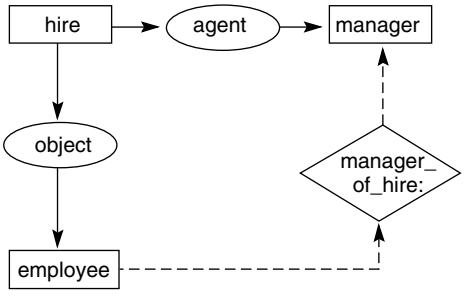


Figure 15.18 Knowledge base entry for hire queries.

Although this example is simplified, it illustrates the use of a knowledge-based approach to building a natural language database front end. The ideas in our example are expressed in conceptual graphs but could equally well be mapped into other representations such as frames or predicate logic-based formalisms.

15.5.3 An Information Extraction and Summarization System for the Web

The World Wide Web offers many exciting challenges as well as interesting opportunities for artificial intelligence and natural language understanding research. One of the biggest is the need to design intelligent software that can summarize "interesting" web-based materials. There are actually two parts to this problem: locating potentially interesting information on the web and then extracting (or mining) critical components of that information. Although the first problem of locating potentially interesting information is

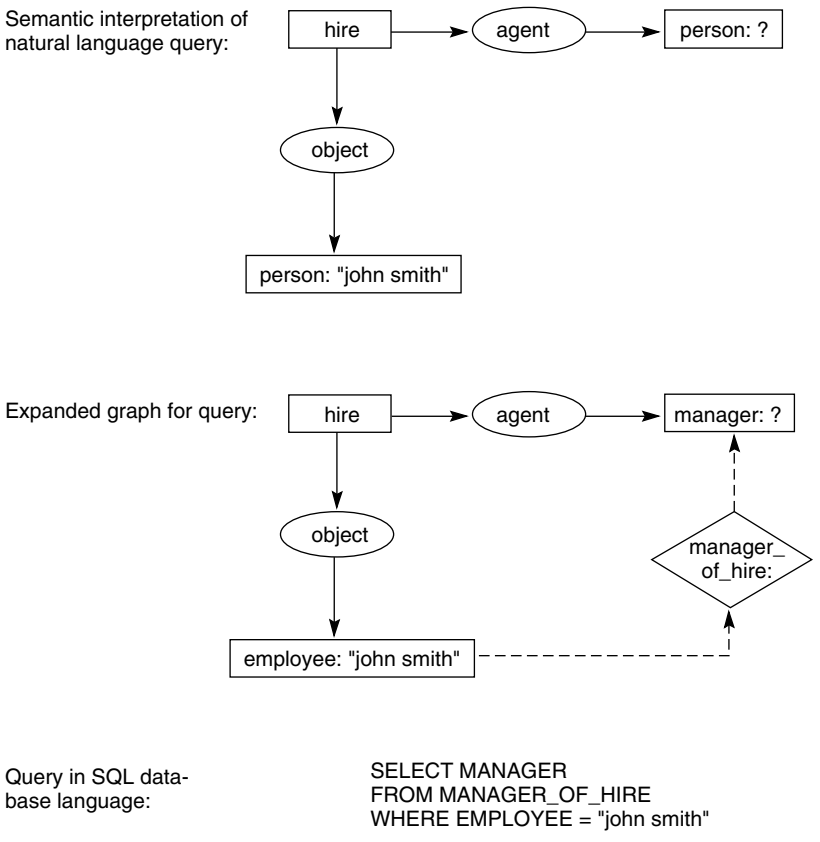


Figure 15.19 Development of a database query from the graph of a natural language input.

critical (and many of the techniques of Chapters 10, 13, and 15 are important), we now consider a template matching and filling approach to address the second problem.

After locating information, perhaps by key-word match or use of a more sophisticated search-engine, an *information extraction system* takes as input this unrestricted text and then summarizes it with respect to a pre-specified domain or topic of interest. It finds useful information about the domain and encodes this information form suitable for report to the user or for populating a structured database.

In contrast to an in-depth natural language understanding system, information extraction systems skim a text to find relevant sections and then focus only on processing these sections. We propose a template-based approach similar to extraction techniques proposed by several researchers (Cardie 1997, Cardie and Mooney 1999). An overview of our information extraction system is presented in Figures 15.20 and 15.21. Suppose we want to find and summarize from the WWW information related to computer science university-level faculty positions. Figure 15.20 presents a target job ad and also presents a template of the information we want our software to extract from this and similar job ads

Sample Computer Science Job Ad (an excerpt):

The Department of Computer Science of the University of New Mexico. . . is conducting a search to fill two tenure-track positions. We are interested in hiring people with research interests in:

Software, including analysis, design, and development tools. . .

Systems, including architecture, compilers, networks. . .

...

Candidates must have completed a doctorate in. . .

The department has internationally recognized research programs in adaptive computation, artificial intelligence, . . . and enjoys strong research collaborations with the Santa Fe Institute and several national laboratories. ...

Sample Partially Filled Template:

Employer: Department of Computer Science, University of New Mexico
Location City: Albuquerque
Location State: NM 87131
Job Description: Tenure track faculty
Job Qualifications: PhD in . . .
Skills Required: software, systems, . . .
Platform Experience: . . .
About the Employer: (text attached)

Figure 15.20 Sample text, template summary, and information extraction for computer science advertisement.

1. Text:	The Department of Computer Science of the University of New Mexico is conducting a search to fill two tenure track positions. We are interested in filling positions at the Assistant Professor. . .
2.Tokenization and Tagging:	The/det Department/noun of/prep ...
3. Sentence Analysis:	Department/subj is conducting/verb search/obj Computer/noun ...
4. Extraction:	Employer: Department of Computer Science Job Description: Tenure track position ...
5. Merging:	tenure track position = faculty New Mexico = NM ...
6. Template Generation:	As in Figure 15.19

Figure 15.21 An architecture for information extraction, from Cardie (1997).

In early attempts at information extraction, natural language systems varied in their approaches. On one extreme, systems processed text using traditional tools: a full syntactic breakdown of each sentence which was accompanied by a detailed semantic analysis. Discourse level processing often followed. At the other extreme, systems used keyword matching techniques with little or no knowledge or linguistic-level analysis. As more systems were built and evaluated, however, the limitations of these extreme approaches to information extraction became obvious. A more modern approach to information extraction, adapted from Cardie (1997), is reflected in Figures 15.20 and 15.21. Although the details of this architecture may differ across applications, the figures indicate the main functions performed in extraction.

First, each sentence of the "interesting" web site is tokenized and tagged. The stochastic tagger presented in Section 15.4 could be used for this. The sentence analysis stage that follows and performs parsing then produces noun groups, verbs, prepositional phrases and other grammatical constructs. Next the extraction phase finds and labels semantic entities relevant to the extraction topic. In our example, this will be to identify employer name, location of the faculty position, job requirements (education, computing skills, and so on), the time the appointment begins, etc.

The extraction phase is the first entirely domain specific stage of the process. During extraction, the system identifies specific relations among relevant components of the text. In our example, the Department of Computer Science is seen as the employer and the location is seen as University of New Mexico. The merging phase must address issues such as synonym reference and anaphora resolution. Example of synonyms are *tenure-track* position and *faculty* position as well as *New Mexico* and *NM*. Anaphora resolution links the *Department of Computer Science* in the first sentence with *we* that is the subject of the second sentence.

The discourse-level inferences made during merging assist the template generation phase, which determines the number of distinct relationships in the text, maps these extracted pieces of information onto each field of the template, and produces the final output template.

In spite of recent progress, current information extraction systems still have problems. First, the accuracy and robustness of these systems can be improved greatly, as errors in extraction seem to follow from a rather shallow understanding of meaning (semantics) of the input text. Second, building an information extraction system in a new domain can be difficult and time consuming (Cardie 1997). Both of these problems are related to the domain-specific nature of the extraction task. The information extraction process improves if its linguistic knowledge sources are tuned to the particular domain, but manually modifying domain-specific linguistic knowledge is both difficult and error prone.

Nonetheless, a number of interesting applications now exist. Glasgow et al. (1997) built a system to support underwriters in analysis of life insurance applications. Soderland et al. (1995) have build a system to extract symptoms, physical findings, test results, and diagnoses from patient's medical records for insurance processing. There are programs to analyze newspaper articles to find and summarize joint business ventures (MUC-5 1994), systems that automatically classify legal documents (Holowczak and Adam 1997), and programs that extract detailed information from computer job listings (Nahm and Mooney 2000).

15.5.4 Using Learning Algorithms to Generalize Extracted Information

Another application brings together several of the ideas presented in this chapter as well as algorithms from machine learning (Section 10.3). Cardie and Mooney (1999) and Nahm and Mooney (2000) have suggested that information extracted from text may be generalized by machine learning algorithms and the result reused in the information extraction task.

The approach is straightforward. Completed, or even partially filled text summarization templates as seen, for example, in Figure 15.20 are collected from appropriate web sites. The resulting template information is then stored in a relational database where learning algorithms, such as ID3 or C4.5 are used to extract decision trees, which, as seen in Section 9.3, can reflect rule relationships implicit in the data sets. (This technique we referred to as *data mining*.)

Mooney and his colleagues propose that these newly discovered relationships might then be used for refining the original templates and knowledge structures used in the information extraction. Examples of this type information that could be discovered from the computer science job application analysis of Section 15.5.3 might include: if the position is computer science faculty then experience on a particular computing platform is not required; if universities are hiring faculty members then research experience is required, etc. Further details on this approach to information generalization may be found in Nahm and Mooney (2000).

There are a large number of current exciting issues in computational linguistics that it is beyond our goal to address. We conclude with three open issues. First, what is the key unit for understanding spoken language? There have been various attempts to answer this question from the full parsing of the English language sentence, attempts at single word recognition, to focus on individual phones. Might it be that focus on the syllable is key to understanding human language? How much of a word do we have to hear before it is recognized? What is the optimum granularity for analysis of voiced speech (Greenberg 1999)?

Second, can access to concepts in a knowledge base through sound recognition lead to further conditioning for subsequent language interpretation? Suppose that the syllable turns out to be a useful unit for language analysis. Suppose that syllable bigrams are useful for suggesting the words of a speaker. Can these words be mapped into the concepts of a knowledge base and the related knowledge be used to enhance the interpretation of subsequent syllables? A caller to a computer based travel organizer could recognize the syllables suggesting a city name and the computer could then anticipate questions related to visiting that city.

Finally, and a long term goal for natural language processing, is the creation of the semantic web; how will this happen? Or will it happen? Is there any methodology under which computers can be said to "understand" anything? Or will there always be search-based pattern-driven "tricks" that are good enough to "convince" people that the computer understands (the Turing-test again)? Isn't this "good enough" approach what we humans do for each other anyway? The final chapter continues this and related discussions.

15.6 Epilogue and References

As this chapter suggests, there are a number of approaches to defining grammars and parsing sentences in natural language. We have presented ATN parsers and Markov models as typical examples of these approaches. The reader should be aware of other possibilities. These include *transformational grammars, semantic grammars, case grammars*, and *feature and function grammars* (Winograd 1983, Allen 1995, Jurafsky and Martin 2008).

Transformational grammars use context-free rules to represent the *deep structure*, or meaning, of the sentence. This deep structure may be represented as a parse tree that not only consists of terminals and nonterminals but also includes a set of symbols called *grammatical markers*. These grammatical markers represent such features as number, tense, and other context-sensitive aspects of linguistic structure. Next, a higher-level set of rules called transformational rules transform between this deep structure and a *surface structure*, which is closer to the actual form the sentence will have. For example, "Tom likes Jane" and "Jane is liked by Tom" have the same deep structure but different surface structures.

Transformational rules act on parse trees themselves, performing the checks that require global context and produce a suitable surface structure. For example, a transformational rule may check that the number feature of the node representing the subject of a sentence is the same as the number feature of the verb node. Transformational rules may also map a single deep structure into alternative surface structures, such as changing active to passive voice or forming an assertion into a question. Although transformational grammars are not discussed in this text, they are an important alternative to augmented phrase structure grammars.

Terry Winograd offers a comprehensive treatment of grammars and parsing in *Language as a Cognitive Process* (1983). This book offers a thorough treatment of transformational grammars. *Natural Language Understanding* by James Allen (1987, 1995) provides an overview of the design and implementation of natural language understanding programs. *Introduction to Natural Language Processing* by Mary Dee Harris (1985) is another general text on natural language expanding the issues raised in this chapter. We also recommend Gerald Gazdar and Chris Mellish (1989), *Natural Language Processing in PROLOG*. Charniak (1993) and Charniak et al. (1993), address issues in stochastic approaches to language and part-of-speech tagging.

The semantic analysis of natural language involves a number of difficult issues that are addressed in knowledge representation (Chapter 7). In *Computational Semantics*, Charniak and Wilks (1976) have articles addressing these issues. Because of the difficulty in modeling the knowledge and social context required for natural language interaction, many authors have questioned the possibility of moving this technology beyond constrained domains. Early research in discourse analysis may be found in Linde (1974), Grosz (1977) and Grosz and Sidner (1990 *Understanding Computers and Cognition* by Winograd and Flores (1986), *Minds, Brains, and Programs* by John Searle (1980) and *On the Origin of Objects* (Smith 1996) address these issues.

Inside Computer Understanding by Schank and Riesbeck (1981) discusses natural language understanding using conceptual dependency technology. *Scripts, Plans, Goals*

and Understanding by Schank and Abelson (1977) discusses the role of higher-level knowledge organization structures in natural language programs.

Speech Acts by John Searle (1969) discusses the role of pragmatics and contextual knowledge in modeling discourse. Fass and Wilks (1983) have proposed *semantic preference theory* as a vehicle for modeling natural language semantics. Semantic preference is a generalization of case grammars that allows transformations on case frames. This provides greater flexibility in representing semantics and allows the representation of such concepts as metaphor and analogy. For a full discussion of the Chomsky hierarchy see Hopcroft and Ullman (1979). We are indebted to John Sowa (1984) for our treatment of conceptual graphs.

Recent presentations of language processing techniques can be found in *Speech and Language Processing* by Jurafsky and Martin (2008), *Foundations of Statistical Natural Language Processing* by Manning and Schutze (1999), and *Survey of the State of the Art in Human Language Technology*, Cole (1997), and the Winter 1997 edition of the *AI Magazine*.

There are a number of very interesting commercial products now available for various aspects of computer based language understanding and generation as well as language based data mining. It is very nice to experiment with these products, but because they are usually not open source, we do not discuss them here.

Excellent references for keeping up with the research trends in natural language understanding, both from the traditional as well as from the stochastic viewpoints, are the annual proceedings of the AI conferences: *AAAI* and *IJCAI*, published by AAAI Press through MIT Press. The *Association of Computational Linguistics* also has its annual national and international conferences as well as the *Journal of the Association for Computational Linguistics.* These are important sources of current technology.

15.7 Exercises

1. Classify each of the following sentences as either syntactically incorrect, syntactically correct but meaningless, meaningful but untrue, or true. Where in the understanding process is each of these problems detected?

   Colorless green ideas sleep furiously.
   Fruit flies like a banana.
   Dogs the bite man a.
   George Washington was the fifth president of the USA.
   This exercise is easy.
   I want to be under the sea in an octopus's garden in the shade.

2. Discuss the representational structures and knowledge necessary to understand the following sentences.

   The brown dog ate the bone.
   Attach the large wheel to the axle with the hex nut.
   Mary watered the plants.
   The spirit is willing but the flesh is weak.
   My kingdom for a horse!

3. Parse each of these sentences using the "dogs world" grammar of Section 15.2.1. Which of these are illegal sentences? Why?

   The dog bites the dog.
   The big dog bites the man.
   Emma likes the boy.
   The man likes.
   Bite the man.

4. Extend the dogs world grammar so it will include the illegal sentences in Exercise 3.

5. Parse each of these sentences using the context-sensitive grammar of Section 15.2.3.

   The men like the dog.
   The dog bites the man.

6. Produce a parse tree for each of the following sentences. You will have to extend our simple grammars with more complex linguistic constructs such as adverbs, adjectives, and prepositional phrases. If a sentence has more than one parsing, diagram all of them and explain the semantic information that would be used to choose a parsing.

   Time flies like an arrow but fruit flies like a banana.
   Tom gave the big, red book to Mary on Tuesday.
   Reasoning is an art and not a science.
   To err is human, to forgive divine.

7. Extend the dogs world grammar to include adjectives in noun phrases. Be sure to allow an indeterminate number of adjectives. Hint: use a recursive rule, adjective_list, that either is empty or contains an adjective followed by an adjective list. Map this grammar into transition networks.

8. Add the following context-free grammar rules to the dogs world grammar of Section 15.2.1. Map the resulting grammar into transition networks.

   sentence ↔ noun_phrase verb_phrase prepositional_phrase
   prepositional_phrase ↔ preposition noun_phrase
   preposition ↔ with
   preposition ↔ to
   preposition ↔ on

9. Define an ATN parser for the dogs world grammar with adjectives (Exercise 7) and prepositional phrases (Exercise 8).

10. Define concepts and relations in conceptual graphs needed to represent the meaning of the grammar of Exercise 9. Define the procedures for building a semantic representation from the parse tree.

11. Extend the context-sensitive grammar of Section 15.2.3 to test for semantic agreement between the subject and verb. Specifically, men should not bite dogs, although dogs can either like or bite men. Perform a similar modification to the ATN grammar.

12. Expand the ATN grammar of Section 15.2.4 to include *who* and *what* questions.

13. Describe how the Markov models of Section 15.4 might be combined with the more symbolic approach to understanding language of Section 15.1–15.3. Describe how the stochastic models could be combined with traditional knowledge-based systems, Chapter 8.

14. Extend the database front end example of Section 15.5.2 so that it will answer questions of the form "How much does Don Morrison earn?" You will need to extend the grammar, the representation language, and the knowledge base.

15. Take the previous problem and put its words, including punctuation, in random order.

16. Assume that managers are listed in the employee_salary relation with other employees in the example of Section 15.5.2. Extend the example so that it will handle queries such as "Find any employee that earns more than his or her manager."

17. How might the stochastic approaches of Section 15.4 be combined with the techniques for database analysis found in Section 15.5.2.

18. Use of the stochastic approach for discovering patterns in a relational database is an important area of current research, sometime referred to as *data mining* (see Section 10.3). How might this work be used to answer queries, such as those posed in Section 15.5.2 about relational databases?

19. As a project, build an information extractions system for some knowledge domain to be used on the WWW. See Section 15.5.3 for suggestions.

20. From this book's auxiliary materials, take the structure of the Prolog context free and context sensitive parsers and ad adjectives, adverbs, and propositional phrases to the grammar.

21. From this book's auxiliary materials, take the structure of the Prolog probabilistic context free parser and ad adjectives, adverbs, and propositional phrases to the grammar.